Todo sobre sistemas embebidos

Arquitectura, programación y diseño de aplicaciones prácticas con el PIC18F

Sergio Salas Arriarán

Todo sobre sistemas embebidos

Arquitectura, programación y diseño de aplicaciones prácticas con el PIC18F

Sergio Salas Arriarán

Lima, setiembre de 2015
Universidad Peruana de Ciencias Aplicadas

© Universidad Peruana de Ciencias Aplicadas (UPC)
Primera publicación: setiembre de 2015

Edición: Diana Félix
Corrección de estilo: Luigi Battistolo
Diseño de cubierta: Christian Castañeda
Diagramación: Diana Patrón Miñán / Christian Castañeda

Editor del proyecto editorial
Universidad Peruana de Ciencias Aplicadas S. A. C.
Av. Alonso de Molina 1611, Lima 33 (Perú) Teléf:
313-3333
www.upc.edu.pe
Primera edición: setiembre de 2015

Versión ebook 2015
Digitalizado y Distribuido por Saxo.com Perú S.A.C.

www.saxo.com/es
yopublico.saxo.com
Telf: 51-1-221-9998
Dirección: Av. 2 de Mayo 534 Of. 304, Miraflores
Lima-Perú

Universidad Peruana de Ciencias Aplicadas (UPC)
Centro de Información

Salas Arriarán, Sergio. Todo sobre sistemas embebidos. Arquitectura, programación y diseño de aplicaciones prácticas con el PIC18F
Lima: Universidad Peruana de Ciencias Aplicadas (UPC), 2015
ISBN de la versión impresa: 978-612-318-033-1
ISBN de la versión pdf: 978-612-318-034-8

Conceptos básicos - Arquitectura del microcontrolador PIC18F - El compilador para PIC18F. El MPLAB X IDE - El lenguaje ensamblador del PIC18F - Rutinas típicas en lenguaje ensamblador - Interrupciones - Los puertos de entrada y salida - El lenguaje ANSI C - Los módulos de temporización - El convertidor analógico digital - El módulo mejorado de comunicación serial síncrona asíncrona (EUSART) - La interfaz serial periférica - La interfaz serial I2C - El módulo USB.

006.22 SALA

Todos los derechos reservados. Esta publicación no puede ser reproducida, ni en todo ni en parte, ni registrada en o transmitida por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea mecánico, fotoquímico, electrónico, magnético, electroóptico, por fotocopia o cualquier otro, sin el permiso previo, por escrito, de la editorial.

El contenido de este libro es responsabilidad del autor y no refleja necesariamente la opinión de los editores.

A mis alumnos, quienes con su interés, a través de constantes preguntas e inquietudes, me animaron al desarrollo de esta obra.

A mis excolegas del INICTEL-UNI, por su compañerismo y las experiencias compartidas.

A mis padres, por las enseñanzas a lo largo de toda la vida.

Contenido

Prólogo **11**

Introducción **13**

Capítulo 1. Conceptos básicos **17**

1.1 Definiciones básicas 17

1.2 Sistemas embebidos 38

1.3 Lenguajes de programación 39

1.4 La familia PIC18F4X de Microchip 42

1.5 Herramientas de desarrollo para Microchip 44

1.6 Características específicas del PIC18F4550 48

Capítulo 2. Arquitectura del microcontrolador PIC18F **51**

2.1 La Unidad central de proceso (CPU) 53

2.2 Las unidades de memoria 59

2.3 Almacenamiento de las instrucciones en la memoria de programa 71

2.4 Las interfaces de entrada y salida (E/S) 78

2.5 El Reset 81

2.6 Temporización y opciones de reloj 84

2.7 Modos de ahorro de energía 88

2.8 El Perro guardián 90

Capítulo 3. El compilador para PIC18F. El MPLAB X IDE **93**

3.1 IDE 93

3.2 Creación de un nuevo proyecto en MPLAB X usando el compilador MPASM 94

3.3 Elaboración de un programa sencillo. Partes del código y reglas básicas 98

3.4. Simulación de un programa en lenguaje ensamblador 106

3.5 Los archivos de configuración 111

3.6 El compilador MPLAB XC8 113

3.7 Creación de un nuevo proyecto en MPLAB usando el compilador MPLAB XC8 115
3.8 Proceso de grabación del microcontrolador PIC18F4550 118

Capítulo 4. El lenguaje ensamblador del PIC18F 123
4.1 El conjunto de instrucciones 124
4.2 Modos de direccionamiento 165
4.3 El Puntero de programa 173
4.4 La pila de direcciones 175
4.5 Instrucciones para el manejo de tablas en la memoria de programa 180
4.6 Acceso a la memoria EEPROM de datos 185
4.7 Directivas para el lenguaje ensamblador 187

Capítulo 5. Rutinas típicas en lenguaje ensamblador 217
5.1 Diseño de un diagrama de flujo 217
5.2 Rutinas básicas de retardo de tiempo 221
5.3 Conversión de bases 225
5.4 Operaciones matemáticas 237
5.5 Detección de teclas pulsadas en un teclado matricial 248

Capítulo 6. Interrupciones 261
6.1 El concepto de interrupción 261
6.2 El vector de interrupción 265
6.3 La prioridad de una interrupción 267
6.4 Latencia de una interrupción 270
6.5 Ejemplo de configuración de una interrupción externa 271

Capítulo 7. Los puertos de entrada y salida 277
7.1 Los puertos de entrada y salida del PIC18F4550 277
7.2 Ejemplo de control de un teclado matricial mediante la interrupción de cambio de estado del Puerto B 282
7.3 Manejo de un módulo LCD alfanumérico basado en el controlador HD44780 286

Capítulo 8. El lenguaje ANSI C **311**

8.1 Historia del lenguaje ANSI C 311

8.2 La estructura de un programa en lenguaje C 315

8.3 Los tipos de datos 317

8.4 Los operadores en ANSI C 322

8.5 Sentencias condicionales e iterativas 328

8.6 Funciones y librerías 338

8.7 Arreglos y punteros 346

8.8 Estructuras y uniones 350

8.9 Tipos de variables compuestos 358

8.10 Uso del lenguaje ensamblador dentro de ANSI C 360

8.11 Manejo de interrupciones en MPLAB XC8 362

8.12 Las directivas del preprocesador 365

8.13 Ejemplos de aplicaciones con lenguaje ANSI C 371

Capítulo 9. Los módulos de temporización **401**

9.1 Los temporizadores del PIC18F4550 401

9.2 Diseño de un semáforo sincronizado con el ***Timer*** 0 414

9.3 Diseño de un reloj en tiempo real con el ***Timer*** 1 420

9.4 El módulo de entrada de captura 423

9.5. El modo de comparación de salida 435

9.6 El modo de modulación por ancho de pulso (PWM) 444

Capítulo 10. El convertidor analógico digital **451**

10.1 Fundamentos de un sistema de adquisición de datos 451

10.2 La conversión analógica digital 454

10.3 El teorema del muestreo 459

10.4 El convertidor analógico digital del PIC18F4550 460

10.5 Ejemplos de aplicaciones con el convertidor analógico digital del PIC18F4550 468

Capítulo 11. El módulo mejorado de comunicación serial síncrona asíncrona (EUSART) **479**

11.1 Nociones básicas de la comunicación serial 479

11.2 La tasa de bit y el formato NRZ 482

11.3 El estándar RS232 484

11.4 El módulo EUSART del PIC18F4550 486

11.5 Ejemplos de aplicación con el EUSART del PIC18F4550 494

Capítulo 12. La interfaz serial periférica **515**

12.1 Fundamentos de la comunicación serial síncrona SPI 515

12.2 Pines del SPI 516

12.3 Formatos de transmisión del bus SPI 517

12.4 El módulo SPI del PIC18F4550 519

12.5 Ejemplos de aplicación con el bus SPI del PIC18F4550 524

Capítulo 13. La interfaz serial I^2C **551**

13.1 Características del protocolo I^2C 551

13.2 Transferencia de datos en el protocolo I^2C 553

13.3 Arbitrariedad 555

13.4 Direccionamiento de periféricos 555

13.5 Formato de transferencia de datos 556

13.6 Registros de configuración del PIC18F4550 557

13.7 Programación del módulo MSSP en modo I^2C 561

13.8 Ejemplos de aplicación con el módulo I^2C del PIC18F4550 564

Capítulo 14. El módulo USB **575**

14.1 Fundamentos del bus USB 576

14.2 Modos de transferencia 581

14.3 El proceso de enumeración 584

14.4 Las clases de descriptores 585

14.5 El módulo USB del PIC18F4550 587

14.6 La librería MLA (***Microchip Library Application***) para dispositivos USB de Microchip 590

14.7 Ejemplo de comunicación entre el PIC18F4550 y un computador a través del puerto USB 593

Bibliografía **603**

Foto: Milagros Segura Zurita

Sergio Salas Arriarán es ingeniero electrónico de la Universidad Peruana de Ciencias Aplicadas y Magíster en Ingeniería Biomédica por la Pontificia Universidad Católica del Perú (PUCP). Se ha desempeñado como Investigador y Desarrollador Tecnológico en el INICTEL-UNI. Ha ejercido la docencia en la Universidad de San Martín de Porres, la Universidad Tecnológica del Perú y la Escuela Naval. Asimismo, cuenta con experiencia como catedrático en la Maestría en Ciencias con mención en Telecomunicaciones de la Universidad Nacional de Ingeniería (UNI). Actualmente, es Profesor a Tiempo Completo de la escuela de Ingeniería Electrónica de la Universidad Peruana de Ciencias Aplicadas (UPC).

Prólogo

Escribir un libro académico es dejar una huella para siempre. Decidir hacerlo no solamente conlleva capacidad, conocimientos, experiencia, dedicación y sacrificio, sino también la intención de que quienes lo lean y lo utilicen puedan aprender su contenido de una manera más fácil y comprensible. Ello constituye un gran desafío, que únicamente puede tener éxito cuando el autor domina el tema y cuando el estilo utilizado para transmitir el conocimiento es muy motivador, utilizando adecuadamente las herramientas audiovisuales disponibles.

En el presente libro, titulado *Todo sobre sistemas embebidos*, Sergio Salas, su autor, ha logrado de manera efectiva compendiar, en sus catorce capítulos, todos los elementos necesarios para la cabal comprensión de dichos sistemas y de sus aplicaciones, a través de ejemplos prácticos. La ruta propuesta para ello comienza por tratar los conceptos básicos acerca de los sistemas de numeración utilizados en la electrónica digital, lo que le sirve de soporte para poder explicar la arquitectura del microcontrolador PIC18F, el cual ha sido elegido en el texto por ser uno de los más populares en la actualidad. Contando con dicha base, procede a explicar el compilador MPLAB X IDE, así como el lenguaje ensamblador con sus rutinas típicas. Introduce luego el concepto de interrupciones en un microcontrolador, y a continuación describe en forma detallada los puertos de entrada y salida del microcontrolador PIC18F4550. Luego justifica la utilidad del lenguaje ANSI C, y lo explica mostrando en todo momento aplicaciones prácticas actuales. Cumplido lo anterior, trata el importante tema de la temporización y sus elementos. A esta altura del camino, el punto siguiente sobre la conversión analógica digital permite apreciar cómo adquirir datos a partir de un sensor y cómo convertirlas de su formato analógico original a uno digital, lo cual hace posible el desarrollo de soluciones para aplicaciones específicas que se puedan necesitar de acuerdo con la identificación de un problema real. Como el microcontrolador debe intercambiar datos con el mundo exterior, el texto completa su esquema desarrollando el módulo mejorado de comunicación serial asíncrona (EUSART), el protocolo de comunicación serial síncrono half dúplex SPI (Serial Peripheral Interface), y, por último, el módulo USB (Universal Serial Bus). A lo largo de cada capítulo, es de destacar la exposición clara y pedagógica, apoyada en la utilización de videos y en preguntas de repaso que, sin duda, distinguen este libro de otros disponibles en la literatura, y que además posibilita y pone en práctica uno de los paradigmas del modelo educativo de la Universidad Peruana de Ciencias Aplicadas (UPC): el de adquirir nuevos conocimientos haciendo las cosas por uno mismo.

El libro está dirigido a estudiantes de pregrado, y es directamente aplicable al curso Sistemas Embebidos de la UPC, ya sea de las carreras de Ingeniería Electrónica, Ingeniería de Telecomunicaciones y Redes, e Ingeniería Mecatrónica. Pero también genera interés en carreras afines, así como en ingenieros que deseen introducirse o actualizarse en este importante tema que, dicho sea de paso, nos encamina al fascinante mundo de internet de las cosas, donde tendremos cada vez más aplicaciones y soluciones para todo tipo de problemas en todos los escenarios que podamos imaginar.

Sin duda alguna, esta obra constituye un valioso aporte para el diseño y la implementación de aplicaciones innovadoras en todos los proyectos de ingeniería que incluyan sistemas embebidos, y contribuye de manera eficaz al desarrollo tecnológico propio que tanto se requiere para la mejora del bienestar y de la competitividad del país.

Carlos R. Valdez Velásquez-López, Dr. Eng.
Director de la Escuela de Ingeniería Electrónica
Universidad Peruana de Ciencias Aplicadas (UPC)

Introducción

En la enseñanza de sistemas embebidos, tanto para estudiantes universitarios como para alumnos a nivel técnico, no se encuentran materiales bibliográficos que involucren temas de arquitectura de sistemas embebidos, lenguajes de programación en ensamblador y ANSI C, que expliquen el funcionamiento y las características de los módulos periféricos de un microcontrolador y que muestren aplicaciones reales basadas en estos dispositivos. Todos estos temas se encuentran de forma separada y en su mayoría, la bibliografía está en idioma inglés. Por esta razón, se decidió tomar todo el material y la experiencia utilizada para la enseñanza de un curso sobre sistemas embebidos y volcarlos en un libro de texto que presente esta información de forma ordenada y simple.

Este libro se ha desarrollado pensando en estudiantes que se introducen al mundo de los sistemas embebidos por primera vez, o que cuentan también con cierta experiencia y desean profundizar conceptos para conocer nuevos ejemplos de aplicaciones con microcontroladores. De la misma manera, el libro está orientado a ingenieros que tengan cierta experiencia en el desarrollo de *hardware* digital o a docentes en el campo de sistemas embebidos que deseen ingresar al mundo de los microcontroladores de la familia PIC18F de Microchip.

Se ha tomado como base para el desarrollo del libro el curso de microcontroladores que se ofrece en las carreras de Ingeniería Electrónica, Ingeniería de Telecomunicaciones y Redes e Ingeniería Mecatrónica de la Facultad de Ingeniería de la Universidad Peruana de Ciencias Aplicadas de Lima, Perú.

Para mostrar ejemplos aplicados se utiliza el PIC18F4550, ya que es uno de los microcontroladores más populares de la empresa Microchip, se encuentra fácilmente en el mercado nacional a un costo bastante accesible y permite desarrollar un sinnúmero de aplicaciones prácticas para soluciones de problemas reales, ya sea en la industria electrónica de consumo o electrónica médica, entre otras. Todos los ejemplos mostrados en el libro se basan en el microcontrolador propuesto. Sin embargo, la gran mayoría de estos se pueden migrar fácilmente a cualquier otro modelo de microcontrolador de la misma familia, ya que los lenguajes de programación utilizados y la arquitectura son compatibles.

Adicionalmente, el PIC18F4550 es uno de los pocos microcontroladores de la familia PIC18F que cuenta con un módulo USB interno, el cual permite que sea conectado al computador a través de este protocolo a velocidades de 1.5Mpbs y 12Mbps, lo que abre las puertas a una serie de aplicaciones relacionadas a interfaces de comunicación con el computador. Es por esta razón que el libro contiene un capítulo dedicado a este tema tan interesante, además de un ejemplo real.

El presente libro pretende ser lo más claro y explicativo posible, a fin de que el lector pueda obtener el máximo provecho. De esta manera, en el aula los alumnos tendrán un mejor entendimiento sobre los temas a tratar en el curso sobre sistemas embebidos, y podrán convertir una sesión de clase explicativa en un debate entre el profesor y los estudiantes, lo cual enriquecerá el nivel de los proyectos propuestos que se puedan producir como resultado de la asignatura.

Cada capítulo del libro trata un tema diferente, pero a la vez necesario para poder entender los capítulos siguientes. Los ejemplos que presentan están listos para que el lector los pueda utilizar directamente en la programación del sistema embebido. Todos estos programas han sido probados en módulos de desarrollo y su funcionamiento se encuentra completamente verificado. El contenido de cada capítulo ha sido dividido de tal manera que el lector primero aprenda a programar en lenguaje ensamblador, entendiendo la arquitectura del microcontrolador y sus modos de acceso a memoria y uso de periféricos de entrada y salida. La segunda parte explica el lenguaje ANSI C, con lo cual algunos ejemplos suben en complejidad, en virtud de la facilidad que conlleva el uso de un lenguaje de alto nivel.

En el primer capítulo se presentan aspectos básicos de los sistemas de numeración binaria, conversión entre bases, las operaciones aritméticas y lógicas en el sistema binario y la representación de cifras decimales, BCD y ASCII. Luego, se explica claramente el concepto de sistema embebido y las herramientas que ofrece Microchip para poder programar y depurar los microcontroladores de la familia PIC18F y otros más. Finalmente, se analizan los elementos más importantes de esta familia y se detallan las características particulares del microcontrolador modelo de este libro: El PIC18F4550.

Posteriormente, el segundo capítulo muestra los detalles de la arquitectura del PIC18F. Se presentan las características del microprocesador, sus registros internos y unidad aritmética lógica. Luego, se explican las distintas unidades de memoria que existen en la familia de microcontroladores PIC18F y las interfaces de entrada y salida, las opciones de Reset, los módulos de temporización, las características del funcionamiento del reloj y los modos de bajo consumo.

En el tercer capítulo se muestra la herramienta de desarrollo: El MPLAB X IDE. Se explica con ejemplos cómo crear un nuevo proyecto, editar un programa, simularlo y grabarlo en la memoria FLASH del microcontrolador. Primero se realiza la demostración con el lenguaje ensamblador y luego se hace lo propio con el lenguaje ANSI C. Esta metodología deberá ser usada por el lector para la verificación del resto de ejemplos presentados en el libro.

El cuarto capítulo detalla las instrucciones del lenguaje ensamblador del PIC18F. Cada instrucción se presenta con un ejemplo que ayuda a entender su forma de operación. A continuación se presentan las directivas del lenguaje ensamblador, las cuales también han sido ejemplificadas, con el objetivo de mostrar claramente su función.

En el quinto capítulo se explica la metodología para plantear un programa en lenguaje ensamblador. Se plantean rutinas básicas que son útiles para la mayoría de aplicaciones, como retardo de tiempo, conversión de bases y operaciones matemáticas. Finalmente, se muestra una técnica para detectar las teclas pulsadas de un teclado matricial conectado al microcontrolador a través de uno de sus puertos.

Seguidamente, el sexto capítulo aborda el tema de las interrupciones. Se explica el concepto y luego se analiza la estructura de las interrupciones ofrecidas por el PIC18F. Luego se muestra un ejemplo sencillo en el que se configurará la interrupción externa 0 del PIC18F4550 para detectar flancos generados por un pulsador.

En el séptimo capítulo se presentan los puertos de entrada y salida del PIC18F4550. Mediante estos conceptos, se muestran ejemplos de configuración y control de un teclado matricial y un módulo LCD alfanumérico.

Luego, en el octavo capítulo se aborda la sintaxis completa del lenguaje ANSI C. Se explican sus instrucciones con ejemplos sencillos y se muestra la forma de creación de librerías haciendo uso del compilador XC8. Se mencionan temas más específicos, como el uso de ensamblador dentro del ANSI C, el manejo de interrupciones y el uso de directivas del preprocesador. Finalmente, se muestran programas de ejemplo, aplicados haciendo uso del lenguaje C y de los periféricos del microcontrolador.

En el noveno capítulo se emprende el tema de los módulos de temporización del microcontrolador. Aquí se presenta la teoría y el mecanismo de funcionamiento de un temporizador, además de diversos ejemplos de funcionamiento. Posteriormente, se explican los modos de Entrada de captura, comparación de salida y generador de PWM, los cuales se ejemplificarán con aplicaciones reales que requieren el uso de servomotores y sensores infrarrojos para su puesta a prueba.

El décimo capítulo toca el tema del convertidor analógico digital del PIC18F. Se presenta la teoría básica de digitalización de señales analógicas y algunos aspectos importantes, como el teorema del muestreo. Luego, se explica el módulo ADC del PIC18F4550 y el método de configuración. Finalmente, se muestran dos ejemplos aplicados para medir el voltaje de forma digital y la temperatura ambiente haciendo uso de un sensor LM35.

Más adelante, en el decimoprimer capítulo se aborda el tema de la comunicación serial asíncrona y el estándar RS232. Se analiza el módulo EUSART del PIC18F y su forma de configuración. Luego, se muestran ejemplos aplicados para obtener una comunicación entre el PIC18F4550 y un computador a través del puerto RS232 y otra aplicación médica muy interesante: el diseño de un oxímetro de pulso para medir el nivel de saturación de oxígeno y pulso cardiaco de un paciente.

En el decimosegundo capítulo se presenta la interfaz serial periférica (SPI), la cual corresponde a uno de los módulos de comunicación síncrona del microcontrolador bastante utilizada por muchos dispositivos periféricos en estos días. Se explica el módulo MSSP del microcontrolador y los registros de configuración. Luego, se muestran dos ejemplos de aplicación: un reloj en tiempo real haciendo uso del circuito integrado DS1305 y el diseño de una matriz de LED basada en el circuito integrado MAX6952.

Posteriormente, el decimotercer capítulo trata de la interfaz serial síncrona I^2C. Se explican las características del protocolo y los métodos de intercambio de información entre dos o más dispositivos conectados mediante esta interfaz. Luego, se explica el módulo MSSP del microcontrolador configurado en modo I^2C y el método de configuración. Se analiza la librería "i2c" del MPLAB XC8 y se muestran dos ejemplos aplicados: uno para obtener la temperatura de un sensor digital como el DS1621 y el acceso de lectura y escritura sobre una memoria EEPROM modelo 24LC08.

El último capítulo toca el tema de la interfaz USB. Se explican de forma general los fundamentos del bus, los métodos de transferencia, el proceso de enumeración y las clases de dispositivo. Luego, se analizan las características más relevantes del módulo USB del PIC18F4550 y se presenta la librería MLA de Microchip, que permite manipular el módulo USB del microcontrolador a través de funciones amigables que manejan los detalles de la configuración, facilitando la elaboración de aplicaciones. Al finalizar el capítulo, se muestra un ejemplo de comunicación entre el PIC18F4550 y el computador a través del bus USB usando la clase CDC.

Capítulo 1. Conceptos básicos

Para la correcta comprensión del contenido del presente capítulo y de los capítulos siguientes, es necesario tener en claro ciertos conceptos básicos acerca de los sistemas de numeración utilizados en la electrónica digital (básicamente, el sistema binario y el hexadecimal), los tipos de operaciones matemáticas relacionadas con estas bases numéricas y las diversas interpretaciones que se pueden obtener de las cifras binarias.

1.1 Definiciones básicas

A continuación se presenta una serie de conceptos importantes sobre electrónica digital; los tipos de base numérica binaria, decimal y hexadecimal; los métodos aritméticos para obtener los cambios de base, operaciones lógicas y aritméticas en estos sistemas numéricos; y los formatos de numeración BCD y ASCII, que son sumamente utilizados en las aplicaciones con sistemas embebidos.

1.1.1 El sistema de numeración binario y hexadecimal

En el mundo de los dispositivos lógicos digitales, las operaciones realizadas siempre involucran patrones numéricos en formato binario. La unidad mínima de este formato es el bit. Un bit se representa por dos niveles o estados lógicos: '1' o '0'. Normalmente, un nivel '1' lógico viene a estar físicamente representado por una magnitud en voltios, y el nivel '0', en otra magnitud de voltaje. Por ejemplo, en la tecnología TTL[1], el nivel lógico '1' se encuentra determinado en el rango entre 2 V y 5.5 V, mientras que el nivel '0' está entre 0V y 0.8 V. Cualquier nivel de voltaje fuera de este rango (entre 0.8 y 2 V) es considerado como un estado indeterminado o 'X', con el cual será imposible realizar operaciones lógicas a nivel digital.

Las cifras binarias, por lo general, no se componen de un solo bit. Normalmente, están compuestas por una cantidad múltiplo de ocho bits. En el gráfico 1.1 se muestran los tipos de arreglos binarios más utilizados en la electrónica digital.

1 *Transistor Transistor Logic*. Tecnología digital basada en transistores bipolares.

Gráfico 1.1. Arreglos binarios más comunes usados en la electrónica digital

Tamaño	Valor mínimo y máximo	Denominación
1 bit	0 – 1	Bit
4 bits	0000 – 1111	Nibble
8 bits	0000 0000 – 1111 1111	Byte
16 bits	0000 0000 0000 0000 – 1111 1111 1111 1111	Word
32 bits	0000 0000 0000 0000 0000 0000 0000 0000 – 1111 1111 1111 1111 1111 1111 1111 1111	Double Word

Elaboración propia.

Una cifra binaria posee la siguiente nomenclatura: (código binario)$_2$, donde el subíndice 2 indica la base. Por ejemplo, la cifra 01010000_2 es un byte con un valor equivalente al número 80 en base decimal (base 10). La nomenclatura para la base hexadecimal se representa antecediendo un 0x sobre el código hexadecimal. Por ejemplo, la cifra mostrada se puede representar como 0x50 en la base hexadecimal. El bit ubicado en el extremo derecho de la cifra en base binaria es conocido como el bit LSB[2] y representa el valor menos significativo. De la misma manera, el bit del extremo izquierdo es considerado el bit MSB[3] o más significativo. Al igual que en el sistema de numeración decimal, una cifra ubicada en la posición de las centenas o millares (izquierda) presenta mayor significancia que una cifra ubicada en la posición de las decenas o unidades (derecha).

En el sistema binario, la base numérica utilizada es 2. Esto significa que cada bit puede estar representado únicamente por dos cifras: '0' o '1'. La posición del bit dentro de la cifra especifica un valor formado por la cifra 2 elevado a una potencia y multiplicado por el valor del bit ('0' o '1'). El valor de dicha potencia es la posición del bit en la cifra. Por ejemplo, si se observa el gráfico 1.2, en el byte de la figura, el bit LSB tiene un aporte de 2^0 (ya que 0 es la posición menos significativa), mientras que el bit MSB aporta con 2^7. Esto implica que el valor de la cifra es el siguiente:

$$valor = 2^7 \times 0 + 2^6 \times 1 + 2^5 \times 0 + 2^4 \times 1 + 2^3 \times 0 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 0$$

Como se puede observar, el valor obtenido es 64+16 = 80, cifra equivalente en base decimal. Conforme se agreguen más dígitos binarios a esta cifra, el exponente de la base binaria se incrementará en uno. Por ejemplo, si una cifra binaria posee 11 bits, el bit MSB tendrá un peso equivalente de 2^{10}.

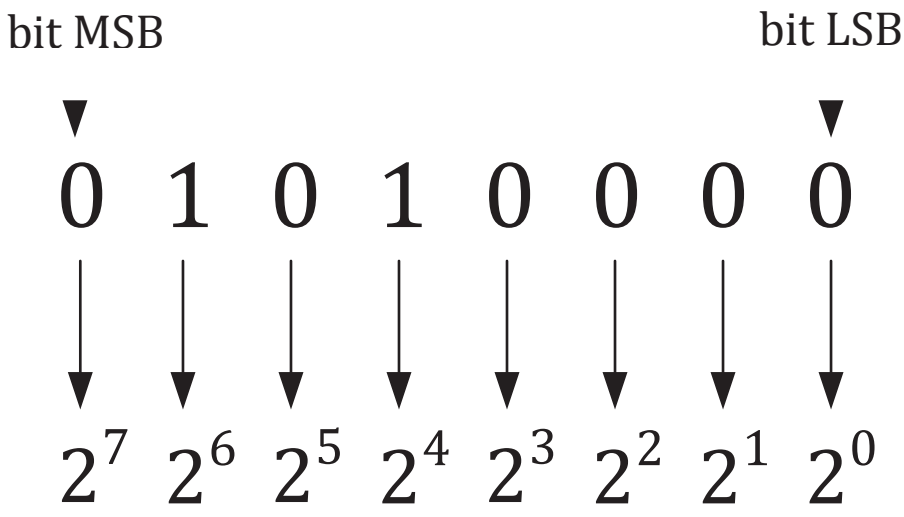
Al igual que en el caso de las cifras decimales, donde existen prefijos como kilo (10^3), Mega (10^6) o Giga (10^9), entre otros que son usados para abreviar magnitudes elevadas, existen también los mismos prefijos para definir grandes magnitudes en cifras binarias formadas por longitudes con base en 8 bits (1 byte). Es así que para el sistema binario:

2 El bit menos significativo. Proviene de las siglas del idioma inglés *Least Significant Bit*.

3 El bit más significativo. Proviene de las siglas del idioma inglés *Most Significant Bit*.

1 kB (Se pronuncia un kilo byte) = 2^{10} bytes o 1024 bytes
1 MB (Se pronuncia un mega byte) = 2^{10} kB o 1024 kB
1 GB (Se pronuncia un giga byte) = 2^{10} MB o 1024 MB

Gráfico 1.2. Ejemplo sobre los pesos de cada bit de una cifra binaria



Elaboración propia.

Al igual que en el sistema de numeración binario, se emplean otros sistemas numéricos para representar cifras en otras bases. Uno de los más usados en la programación de sistemas embebidos es la base hexadecimal[4]. En ella cada cifra tiene un peso, dependiendo de una potencia en base 16. En este sistema existen 16 cifras diferentes (valores de 0 a 15) para representar una magnitud. Por ejemplo, para las cifras del 0 al 9 se utilizan los mismos dígitos que en el sistema decimal. Para completar los 6 dígitos restantes se utilizan letras del alfabeto, como el siguiente caso:

0xA	=	10	=	1010_2
0xB	=	11	=	1011_2
0xC	=	12	=	1100_2
0xD	=	13	=	1101_2
0xE	=	14	=	1110_2
0xF	=	15	=	1111_2

En el número hexadecimal mostrado en el gráfico 1.3, cada cifra representa un *nibble*, dado que se tienen 16 posibles valores diferentes y esto equivale a una combinación de 4 bits (2^4 estados). Cada dígito representa una potencia de 16, según su posición en la cifra. En el ejemplo, el valor decimal equivalente del número hexadecimal mostrado es el siguiente:

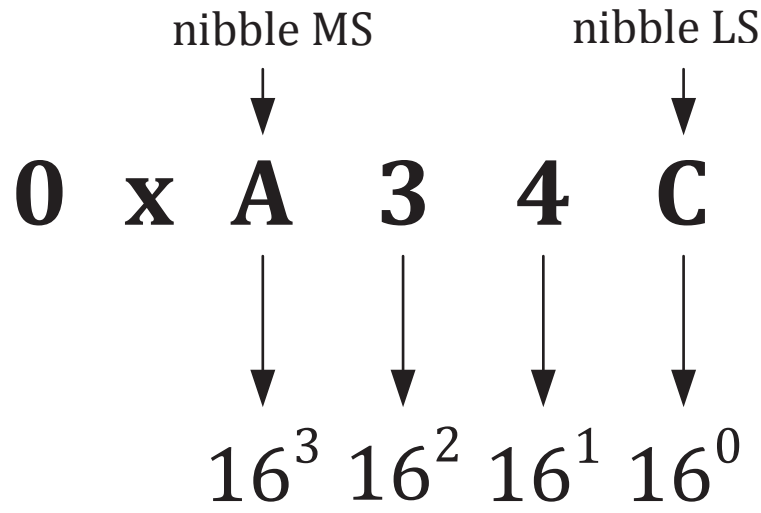
4 Observar que en esta publicación para la notación hexadecimal se utiliza el símbolo «0x» (cero equís). Es distinto del símbolo «por» utilizado la multiplicación.

$$valor = 16^3 \times 10 + 16^2 \times 3 + 16^1 \times 4 + 16^0 \times 12$$

De esta manera, el valor decimal equivalente es 40 960 + 768 + 64 + 12 = 41 804. Como se puede observar, esta cifra pudo haber sido representada en el sistema binario con el siguiente código: 1010001101001100_2, lo cual habría resultado en una cifra más larga, al estar conformada por 16 dígitos. En cambio, en el sistema hexadecimal se utilizan únicamente 4 dígitos para representar la misma cifra numérica, lo cual permite abreviar valores de magnitudes muy extensas.

El sistema de numeración hexadecimal es el preferido cuando se quiere representar magnitudes superiores a 2^8 y, al mismo tiempo, se desea mantener la visualización del código binario. Esto permite obtener una mejor comprensión del orden de la cifra sin perder de vista su forma a nivel binario (ya que es fácilmente visible el código binario a partir de cada *nibble* o dígito hexadecimal).

Gráfico 1.3. Ejemplo sobre los pesos de cada cifra hexadecimal



Elaboración propia.

1.1.2 Conversiones entre bases

La base decimal es el formato más utilizado por los seres humanos para interpretar las cifras numéricas que operan diariamente para sus tareas cotidianas. Las personas comunes y corrientes han aprendido a realizar operaciones aritméticas en base a este sistema de numeración. Entonces, la pregunta que cae por su propio peso es la siguiente: «¿por qué el programador debe conocer otros sistemas de numeración, como el binario y el hexadecimal?» La respuesta a esta pregunta es obvia: porque al programar un sistema embebido no se debe pensar como seres humanos sino como un computador, y el sistema de numeración utilizado por estas máquinas es, por naturaleza, el binario.

Durante la programación de un sistema embebido no es necesario que el programador realice las conversiones de bases en el código ingresado. Para el software de programación de sistemas embebidos, el uso de una base u otra es indiferente. La conversión de una base es un proceso automáticamente efectuado por el compilador en el momento de generar el código de salida del programa. Sin embargo, el programador debe conocer las diferentes bases para poder interpretar las cifras mentalmente y así

agilizar la implementación de su algoritmo. La base decimal es por excelencia la más sencilla de interpretar por cualquier ser humano. Por tal motivo, se toma como referencia en las conversiones de base.

Convertir una cifra de base binaria a decimal es bastante simple y ya se explicó en el subcapítulo 1.1.1. Solo basta con multiplicar cada dígito binario ('0' o '1') por su correspondiente peso (base dos elevado a la posición del dígito) y sumar todos los dígitos resultantes. Por ejemplo, si se tiene la cifra binaria 10101010_2, la transformación a la base decimal se haría de la siguiente manera:

$$valor\ decimal = 2^7 \times 1 + 2^6 \times 0 + 2^5 \times 1 + 2^4 \times 0 + 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0$$

De tal modo, el valor decimal equivalente obtenido es 128 + 32 + 8 + 2 = 170.

Para realizar la conversión de una cifra binaria a hexadecimal, el procedimiento también resulta sumamente sencillo. Si se toma como ejemplo el valor binario 010111000011_2, primero se deberá separar la cifra en el número total de *nibbles* que lo componen. Para este caso son tres: 0101_2, 1100_2 y 0011_2. Cada *nibble* posee su equivalente en cifra hexadecimal. Por ejemplo, 0101_2 es 0x5, 1100_2 es 0xC (doce) y 0011_2 es 0x3. Por lo tanto, la cifra hexadecimal equivalente es 0x5C3.

El proceso de conversión de decimal a binario es ligeramente más complejo que en los casos anteriores. Consiste en realizar divisiones sucesivas entre 2 hasta lograr que el cociente final de la división sea 1. Si se desea convertir el valor decimal 17 a código binario, se deben realizar las siguientes operaciones sucesivas:

17	÷	2	= 8,	Residuo	= 1	(bit LSB, primer dígito)
8	÷	2	= 4,	Residuo	= 0	(segundo dígito)
4	÷	2	= 2,	Residuo	= 0	(tercer dígito)
2	÷	2	= 1 (bit MSB, quinto dígito)	Residuo	= 0	(bit LSB, cuarto dígito)

La cifra binaria equivalente se forma al tomar el último cociente como bit MSB y los residuos restantes como los dígitos ordenados de izquierda a derecha. La cifra binaria, entonces, estará compuesta de 5 dígitos y será el valor 10001_2.

Si se repite el ejercicio con una cifra decimal par, se podrá notar que el bit LSB siempre será '0', ya que la primera división entre 2 de un número par dará como residuo 0. El dígito MSB siempre será '1' para cualquier cifra. Por ejemplo, para convertir el número 18 a base decimal se tiene la siguiente operación:

18	÷	2	= 9,	Residuo	= 0	(bit LSB, primer dígito)
9	÷	2	= 4,	Residuo	= 1	(segundo dígito)
4	÷	2	= 2,	Residuo	= 0	(tercer dígito)
2	÷	2	= 1 (bit MSB, quinto dígito)	Residuo	= 0	(bit LSB, cuarto dígito)

Como resultado se obtiene el código binario 10010_2, del cual se puede observar que el bit LSB es '0'.

La conversión de base hexadecimal a decimal es muy parecida a la de binario a decimal, con la diferencia de que cada elemento tiene asociada una potencia en base 16. Por ejemplo, se tiene un número hexadecimal cuyo valor es 0xFA28. Para convertirlo a su equivalente en base decimal, se debe observar que el número en base 16 estará compuesto por 4 cifras:

0xF = 15 (*nibble* más significativo),
0xA = 10,
0x2 = 2 y
0x8 = 8 (*nibble* menos significativo)

La cifra 0xF es la de mayor significancia (MS[5]), por lo cual le corresponde un peso de 16^3, mientras que la cifra 0x8 es la menos significativa (LS[6]) y, por lo tanto, le corresponde el menor peso, de 16^0. Posteriormente, se multiplica cada cifra por su respectiva potencia de 16 según el orden:

$$valor = 16^3 \times 15 + 16^2 \times 10 + 16^1 \times 2 + 16^0 \times 8$$

De esta manera, el valor decimal da como resultado 61 440 + 2560 + 32 + 8 = 64 040. Cada cifra es múltiplo de 16; sin embargo, es posible expresar cualquier magnitud entera en base hexadecimal.

Para pasar de una base decimal a hexadecimal el procedimiento también resulta ser similar al caso de la conversión decimal a binaria. La diferencia radica en que las divisiones sucesivas se realizan sobre un divisor de 16. Suponiendo que se tiene el número 39 481 en base decimal y se desea transformar a base hexadecimal, el procedimiento sería el siguiente:

39 481	÷	16	= 2467.	Residuo	= 9	(cifra LS, primer dígito)
2467	÷	16	= 154.	Residuo	= 3	(segundo dígito)
154	÷	16	= 9 (cifra MS, cuarto dígito)	Residuo	= 10	(tercer dígito)

El resultado de la conversión de 39 481 a hexadecimal es, entonces, 0x9A39. La división sucesiva deberá detenerse cuando el último cociente obtenido sea un valor menor a 15 (0xF).

Un proceso mucho más sencillo y que no requiere de cálculos aritméticos es la conversión binaria a hexadecimal. Como ejemplo, se cuenta con el número binario 1110011011000010_2 y se desea transformar a base hexadecimal. El primer paso de este proceso es dividir el número binario en grupos de cuatro bits o *nibbles*. Si no se cuenta con un número de bits múltiplo de cuatro, se deberán agregar varios '0' a la izquierda para completar los dígitos. Entonces, el número binario se divide de la siguiente manera:

Nibble MS: 1110_2 = 0xE
Segundo *nibble*: 0110_2 = 0x6
Tercer *nibble*: 1100_2 = 0xC
Nibble LS: 0010_2 = 0x2

5 *Most Significant* (más significativo).

6 *Least Significant* (menos significativo).

Por tanto, la cifra hexadecimal equivalente es 0xE6C2.

La conversión hexadecimal a binario es un proceso muy similar al anterior. Simplemente se deberá tomar cada cifra del número hexadecimal, para convertirla al *nibble* correspondiente en código binario. Por ejemplo, se tiene el número hexadecimal 0xABCD2 y se desea convertirlo a binario. El procedimiento es como se muestra:

Nibble MS: $0xA = 1010_2$
Segundo *Nibble*: $0xB = 1011_2$
Tercer *Nibble*: $0xC = 1100_2$
Cuarto *Nibble*: $0xD = 1101_2$
Nibble LS: $0x2 = 0010_2$

De esta manera, el resultado de la conversión es 10101011110011010010_2. Se puede observar que no es conveniente expresar una cifra de esta magnitud en base binaria, ya que resulta muy larga y difícil de interpretar. Durante la programación de sistemas embebidos, el programador es libre de utilizar cualquiera de las tres bases vistas (base binaria, decimal y hexadecimal). Sin embargo, muchos programadores optan por utilizar la combinación de las tres bases en sus programas, ya que, para ciertas circunstancias, algunas bases se adecúan mejor que otras. Por lo general, un programador opta por la base decimal cuando realiza operaciones matemáticas. La base hexadecimal se utiliza para manejar direcciones de memoria (que usualmente son magnitudes elevadas) o cargar códigos binarios a registros, y la base binaria suele ser utilizada para la asignación de estados a los puertos de entrada y salida (E/S) del sistema embebido.

Si bien es cierto que existen otras bases, como la octal (base 8), estas son poco utilizadas, debido a que no son versátiles y no resultan atractivas para la mayoría de programadores (al parecer, conocer tres bases numéricas ya es suficiente). No obstante, la mayoría de compiladores siguen aceptando este tipo de base.

1.1.3 Operaciones lógicas

En el mundo digital existen dos maneras de operar las cifras binarias (o en la base en que se interpreten, pues finalmente todo número es manejado de forma binaria en un sistema embebido). Cada cifra puede ser operada en modo aritmético o en modo lógico. El modo aritmético se presentará en el siguiente subcapítulo. A continuación se muestra el modo lógico.

El modo lógico se basa en las operaciones lógicas. Estas se obtienen mediante la manipulación de los patrones binarios en base a las reglas creadas por George Boole[7] a mediados del siglo XIX, las cuales constituyen el álgebra de Boole. El álgebra de Boole opera sobre variables que pueden ser verdaderas o falsas. Las reglas de esta álgebra son actualmente utilizadas en las operaciones lógicas binarias en los sistemas digitales, ya que cada cifra binaria puede ser tratada como verdadero (nivel binario '1') y falso (nivel binario '0')[8].

7 Filósofo y matemático inglés, padre de la lógica booleana [1815-1864].

8 Cfr. Sánchez y Canton 2007.

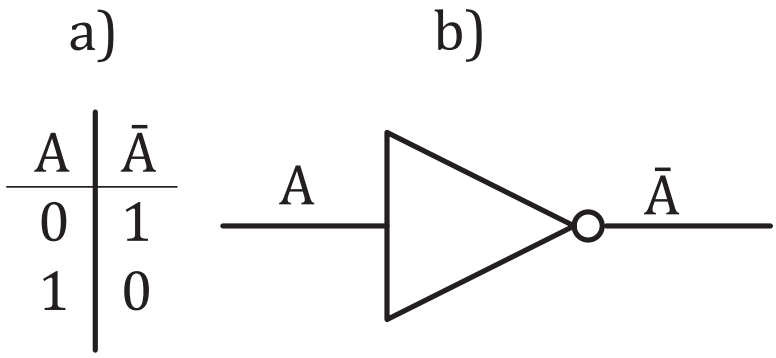
Las operaciones lógicas típicas más empleadas en los sistemas embebidos son las siguientes:

- NOT
- OR
- AND
- XOR
- SWAP
- SHIFT

Cabe señalar que las operaciones lógicas son fundamentales en la programación de sistemas embebidos, ya que permiten la implementación de una serie de operaciones vitales para el funcionamiento del programa. Por ejemplo, invertir el estado de un pin de entrada y salida de un sistema embebido, configurar un registro interno, realizar operaciones de conversión binario BCD o implementar el proceso de división, entre otras. Estos son ejemplos en los cuales se requiere hacer uso intensivo de las operaciones lógicas.

La operación lógica más simple de todas es la NOT (negación). Esta operación simplemente invierte un estado verdadero a falso y viceversa. En el caso binario, negar un bit significa cambiar su estado de '1' a '0' o de '0' a '1'. Por ejemplo, se tiene una variable de nombre A que representa un bit. Por lo tanto, A puede tomar dos valores: '0' o '1'. Si se niega el valor de A, entonces Ā (se denomina a esta variable 'A negado'. En otras literaturas se considera A´ también como 'A negado') denota la negación. El símbolo para la compuerta lógica que realiza la negación se muestra en el gráfico 1.4, junto con su tabla de verdad.

Gráfico 1.4. a) Tabla de verdad de la operación lógica NOT. b) Símbolo de la compuerta NOT



Elaboración propia.

También es posible realizar la operación lógica NOT sobre un conjunto de bits, que podrían ser un *nibble* o un byte. Si la variable A equivale al siguiente valor:

$$A = 10110011_2$$

Entonces:

$$\overline{A} = 01001100_2$$

La operación lógica OR, también conocida como OR inclusiva (IOR), es otra de las operaciones básicas del álgebra de Boole. Su lógica es muy similar a la suma, excepto cuando ambos operandos son '1'[9]. En una operación OR deben existir al menos dos operandos, y basta que uno de ellos sea '1' para que el resultado de la operación sea '1'. En el gráfico 1.5 se observa la tabla de verdad y el símbolo de la compuerta lógica OR.

La operación OR se realiza normalmente en conjuntos binarios. Por ejemplo, en el caso de que la variable A = 10000011_2 y la variable B = 00011111_2, el resultado de la operación OR sería el siguiente:

$$A\ OR\ B = A + B = 10000011_2 + 00011111_2 = 10011111_2$$

También es posible combinar las operaciones. Por ejemplo, se puede negar la operación OR entre A y B, lo cual se conoce como la operación NOR. Esto daría el siguiente resultado para los valores dados de A y B:

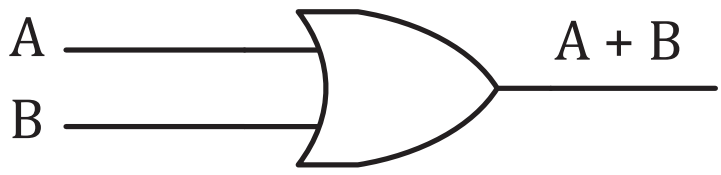
$$\overline{A\ OR\ B} = \overline{A + B} = \overline{10000011_2 + 00011111_2} = \overline{10011111_2} = 01100000_2$$

Gráfico 1.5. a) Tabla de verdad de la operación lógica OR. b) Símbolo de la compuerta OR

a)

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

b)



Elaboración propia.

En muchos sistemas embebidos, la operación NOR no forma parte del conjunto de instrucciones. Por lo cual, para su implementación, se deben ejecutar las operaciones OR y NOT de manera combinada.

La operación lógica AND toma dos variables binarias y realiza un proceso muy similar al de la multiplicación, sin tomar en cuenta el acarreo. En esta operación basta que uno de los operandos sea '0' para que todo el resultado de la operación dé '0'. En el gráfico 1.6 se muestra la tabla de verdad y el símbolo de la compuerta lógica AND.

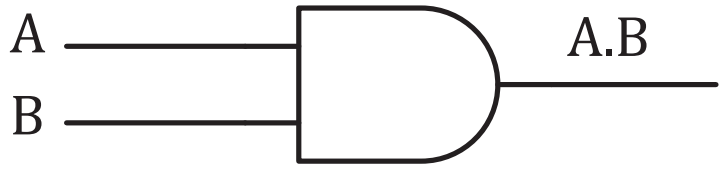
9 Si se tiene la operación 1_2 OR 1_2, el resultado en una suma debería ser 10_2. Sin embargo, para la operación lógica el resultado es 1_2.

Gráfico 1.6. a) Tabla de verdad de la operación lógica AND. b) Símbolo de la compuerta AND

a)

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

b)



Elaboración propia.

En el caso de que la variable A = 11001100_2 y la variable B = 01001011_2, el resultado de la operación AND será el siguiente:

$$A\ AND\ B = A.B = 11001100_2.01001011_2 = 01001000_2$$

En el resultado se puede observar que solo aquellos bits que poseen la cifra '1' en la misma posición mantienen su valor '1'. El resto son '0'. También es posible agregar una tercera variable, de nombre C, a la operación. Si se tiene que C = 00001111_2, entonces la operación AND de las tres variables generará el siguiente resultado:

$$A\ AND\ B\ AND\ C = A.B.C = 11001100_2.01001011_2.00001111_2 = 00001000_2$$

También se puede combinar la operación lógica AND con la NOT para obtener un operador NAND. Este tipo de operación tampoco es muy común en el conjunto de instrucciones de sistemas embebidos con arquitectura tipo RISC[10]. Por ejemplo, la operación NAND de las variables A y B se muestra a continuación:

$$\overline{A\ AND\ B} = \overline{A.B} = \overline{11001100.01001011} = \overline{01001000_2} = 10110111_2$$

Hasta el momento se han analizado las tres operaciones lógicas fundamentales. Si se combinan estos tres operadores, se logrará obtener una nueva operación lógica que se encuentra disponible en la arquitectura de la gran mayoría de sistemas embebidos: el operador XOR (OR exclusivo). Este operador genera un resultado verdadero solo si una de las entradas es verdadera y la otra es falsa. Pero si ambas entradas a la operación XOR son iguales, el resultado será falso.

10 Conjunto de instrucciones reducido, de las siglas en inglés *Reduced Instruction Set Computing*. Es un tipo de arquitectura de sistemas embebidos que se explicará en el segundo capítulo.

Si A y B son dos variables binarias, entonces la operación XOR se denota de la siguiente manera:

$$A\ XOR\ B = A \oplus B = A.\overline{B} + \overline{A}.B$$

En el gráfico 1.7 se muestra la tabla de verdad de la operación lógica XOR y el símbolo de su compuerta.

La operación XOR es muy utilizada para verificar si dos números son iguales. Por ejemplo, la variable A = 10001110_2 y la variable B = 10001110_2. Ambas variables tienen asignados el mismo código binario. Entonces, la operación XOR dará el siguiente resultado:

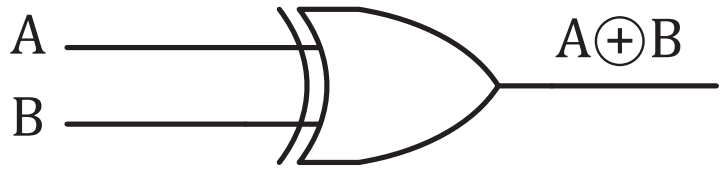
$$A \oplus B = 10001110_2 \oplus 10001110_2 = 00000000_2$$

Gráfico 1.7. a) Tabla de verdad de la operación lógica AND. b) Símbolo de la compuerta XOR

a)

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

b)



Elaboración propia.

Como se puede observar, al ser todos los bits iguales en cada posición, el resultado de la operación XOR da 00000000_2. Este resultado será el mismo cada vez que los operandos sean iguales. Si, en cambio, se alterase un bit de la variable B y su nuevo valor fuera B = 10011110_2, entonces el resultado de la operación XOR sería el siguiente:

$$A \oplus B = 10001110_2 \oplus 10011110_2 = 00010000_2$$

El valor obtenido es diferente de 00000000_2, lo cual será siempre así si es que ambos operandos no son iguales.

La operación SWAP no posee un símbolo lógico. Si bien es cierto, no es una operación lógica propiamente dicha sino el resultado de la combinación de operaciones básicas, muchos sistemas embebidos utilizan una instrucción de nombre *swap* para alterar el contenido de un byte de información.

En la operación *swap*, los *nibbles* de un byte son intercambiados de posición. Con una variable A = 11000101_2, se observa que el *nibble* más significativo es 1100_2, mientras que el *nibble* menos significativo es 0101_2. Si se realiza la operación *swap* sobre A, el nuevo resultado será A = 01011100_2. Como se puede constatar, simplemente se han intercambiado de posición los *nibbles* del byte.

Otra de las operaciones clásicas realizadas por todo sistema embebido es la operación *shift* o de desplazamiento binario. En una cifra binaria, cada bit posee una posición. Con el operador *shift* es posible realizar un cambio de posición de cada bit, ya sea para la derecha o para la izquierda.

Nuevamente, la operación *shift* no es una compuerta, sino un conjunto de operaciones que involucran el movimiento de los bits en un registro. Esta operación está incluida en el conjunto de instrucciones de todo sistema embebido y resulta muy útil para una gama de aplicaciones en las cuales se requiere multiplicar, dividir o convertir de binario a BCD, entre otras.

En la operación *shift* se utilizan los símbolos >> o <<, para denotar un corrimiento binario hacia la derecha o hacia la izquierda, respectivamente. Por ejemplo, si se tiene que la variable A = 10011000_2, entonces, al realizar la operación A>>3, los bits de la variable A se desplazarán tres posiciones a la derecha, quedando de la siguiente manera: A = 00010011_2. Al realizar esta operación han aparecido tres bits de valor cero (b'000') al lado izquierdo del bit que solía ser el MSB de la cifra. Estas tres posiciones en realidad son reemplazadas por bits fuera de las ocho posiciones del byte. Por defecto, cuando se realiza el corrimiento, estos nuevos valores son siempre ceros.

Ahora, en el caso de que la variable A = 00010011_2 sufre un corrimiento a la izquierda con la siguiente operación: A<<4, el resultado quedaría de la siguiente forma: A = 0011000_2. En este caso se puede observar que uno de los bits '1' de la variable A ha desaparecido (el bit en la posición 4). Esto en realidad no ha ocurrido, sino que este bit ha tomado la posición 8 o se ha convertido en el noveno bit, el cual no se muestra, ya que el valor a tratar es un byte y solo tiene 8 posiciones (la novena posición, verdaderamente, solo estará en la mente del programador).

1.1.4 Operaciones aritméticas

Otra de las formas que existen para manipular los números binarios a nivel digital es a través de operaciones aritméticas. Las operaciones aritméticas típicas que involucran a los sistemas embebidos de mediano rango son las siguientes:

- Suma
- Resta
- Multiplicación

Las reglas de la aritmética que se conocen para el sistema decimal son muy similares al sistema binario. La diferencia es que el sistema binario es más simple y las reglas que hay que aprender para efectuar una suma, por ejemplo, son más sencillas que en el caso de la base decimal. En general, para llevar a cabo una sumatoria en el sistema binario se deben tener en cuenta únicamente tres reglas:

$0_2 + 0_2 = 0_2$
$0_2 + 1_2 = 1_2$
$1_2 + 0_2 = 1_2$
$1_2 + 1_2 = 10_2$ (0 y un bit de acarreo)

} La suma aritmética es conmutativa.

El acarreo es una parte fundamental en la suma. Por ejemplo, en la suma decimal es común que se produzca un acarreo cuando la sumatoria supera el valor de nueve. Se empieza sumando las cifras menos

significativas y se termina cuando la última suma involucra a las dos cifras más significativas de los operandos (o a solo uno de ellas). Si se analiza la siguiente suma decimal: 137 + 26, sucederá lo siguiente:

1. 7 + 6 = 3 (acarreo1 = 1)
2. (acarreo1) + 3 + 2 = 6 (acarreo2 = 0)
3. (acarreo2) + 1 = 1

Resultado = 163

En general, para la base decimal el acarreo es un número entre 0 y 1. Para el caso de la base binaria, el acarreo también será una cifra entre 0_2 y 1_2, ya que la suma de dos dígitos binarios solo puede dar '0', '1' o 10_2. En el caso de que se quiera realizar la suma de los siguientes dígitos binarios: 01101100_2 y 10011010_2, primero, se comienza sumando las cifras menos significativas, de la siguiente manera:

1. '0' + '0' = '0' (acarreo1 = '0')
2. (acarreo1) + '0' + '1' = '1' (acarreo2 = '0')
3. (acarreo2) + '1' + '0' = '1' (acarreo3 = '0')
4. (acarreo3) + '1' + '1' = '0' (acarreo4 = '1')
5. (acarreo4) + '0' + '1' = '0' (acarreo5 = '1')
6. (acarreo5) + '1' + '0' = '0' (acarreo6 = '1')
7. (acarreo6) + '1' + '0' = '0' (acarreo7 = '1')
8. (acarreo7) + '0' + '1' = 10_2

Resultado = 100000110_2

Como se puede observar, el resultado posee un bit más (9 bits en total) que los dos operandos. Esto se debe al acarreo que se ha ido arrastrando desde la cuarta operación. En el ejemplo anterior, el primer operando 01101100_2 representa la cifra decimal 108, y el segundo operando 10011010_2 representa la cifra decimal 154. El resultado es 100000110_2, cuyo valor decimal es 262, y corresponde a la suma de 108 + 154.

Uno de los formatos utilizados por los sistemas embebidos para poder operar números positivos y negativos es el complemento a 2. Las cifras binarias pueden interpretarse de dos formas: positivas o negativas. Por ejemplo, la cifra 10010011_2 puede interpretarse de dos maneras: como 147 en base decimal o –109 (complemento a 2).

En el formato complemento a 2, una cifra se puede interpretar como positiva o negativa. En este formato, el bit MSB representa el signo. Si este bit es '1', entonces el número tendrá una representación negativa. Para obtener el valor negativo de una determinada magnitud en código binario, se debe obtener su complemento a 2. Esto se logra realizando los siguientes pasos:

1. Escribir la cifra binaria.
2. Realizar la operación NOT sobre toda la cifra.
3. Sumar el bit 1_2 a la cifra negada.
4. Si la cifra inicial representa una magnitud positiva, el resultado será la representación negativa en complemento a 2. Si la cifra inicial es una magnitud negativa, el resultado será la representación positiva en complemento a 2.

De esta manera, para determinar el valor representado por la cifra 10010011_2 en complemento a 2, el primer paso consiste en invertirla y transformarla en el siguiente valor negado: 01101100_2. La siguiente fase consiste en sumar el bit '1', con lo cual cambia al siguiente estado: 01101101_2. Finalmente, se interpreta esta cifra en magnitud positiva, cuyo valor es 64 + 32 + 8 + 4 + 1 = 109. Entonces, el valor representado por 10010011_2 en complemento a 2 es –109.

Las mismas reglas que se emplean para la adición de números sin signo aplican para los números complemento a 2. Por ejemplo, si se suma –109 + 100, el resultado es –9. Esto, en binario, sería equivalente a obtener la suma de $10010011_2 + 01100100_2 = 11110111_2$. Dicho valor en complemento a 2 es 00001001_2, que equivale a la cifra 9 decimal, lo cual implica que la suma debe interpretarse como –9.

Las reglas de adición pueden combinar cifras sin signo y complemento a 2 sin ninguna distinción. Por ejemplo, en la siguiente sumatoria: (–9) + 8 + 1 + (–4), el resultado es el complemento a 2 de 4. En cifras binarias sería lo siguiente:

$$11110111_2\ (-9) + 00001000_2\ (8) + 00000001_2\ (1) + 11111100_2\ (-4)$$

El procedimiento consiste en sumar cada bit empezando por los bits LSB de cada una de las cuatro cifras:

1. $1_2+0_2+1_2+0_2 = 0_2$ (acarreo1 = 1_2)
2. (acarreo1) + $1_2+0_2+0_2+0_2 = 0_2$ (acarreo2 = 1_2)
3. (acarreo2) + $1_2+0_2+0_2+1_2 = 1_2$ (acarreo3 = 1_2)
4. (acarreo3) + $0_2+1_2+0_2+1_2 = 1_2$ (acarreo4 = 1_2)
5. (acarreo4) + $1_2+0_2+0_2+1_2 = 1_2$ (acarreo5 = 1_2)
6. (acarreo5) + $1_2+0_2+0_2+1_2 = 1_2$ (acarreo6 = 1_2)
7. (acarreo6) + $1_2+0_2+0_2+1_2 = 1_2$ (acarreo7 = 1_2)
8. (acarreo7) + $1_2+0_2+0_2+1_2 = 11_2$

Resultado = 111111100_2

Se puede observar que el resultado presenta 9 bits de longitud. Sin embargo, los bits más significativos son 1_2, lo cual se puede interpretar como ceros a la izquierda o cifras no significantes para una magnitud complemento a 2. Si se niega Resultado y se suma 1, el resultado interpretado será 000000100_2 o –4, lo cual es equivalente al resultado obtenido en base decimal.

Las técnicas utilizadas para restar números en base decimal también son muy similares a las que se aplican para la resta en base binaria. El proceso comienza restando los bits LSB hasta llegar a los bits MSB. En cualquier resta binaria en la que una cantidad mayor debe ser restada de una menor, se presentará un dígito de acarreo, que en este caso es prestado del bit de la posición izquierda. De esta manera, las reglas básicas de sustracción binaria son las siguientes:

'0' – '0' = '0'
^{1}'0' – '1' = '1'
'1' – '0' = '1'
'1' – '1' = '0'

} Se presta un bit '1' del bit más significativo inmediato.

Por ejemplo, si se desea restar 56 – 37 en decimal, se debe realizar el siguiente procedimiento:

1. 6 – 7 = 9 (se presta 1 del dígito más significativo, por lo cual se acarrea 1: acarreo1 = 1)
2. 5 – (acarreo1) – 3 = 1

Resultado = 19

Si luego estas cifras se transforman a base binaria, en donde 56 = 00111000_2 y 37 = 00100101_2, la resta se realizará empezando desde los bits LSB hasta los bits MSB, de la siguiente manera:

1. ^{1}'0' – '1' = '1' (se presta '1' del siguiente bit más significativo, acarreo1 = '1')
2. ^{1}'0' – '0' – acarreo1 = '1' (se presta '1' del siguiente bit más significativo, acarreo2 = '1')
3. ^{1}'0' – '1' – acarreo2 = '0' (se presta '1' del siguiente bit más significativo, acarreo3 = '1')
4. '1' – '0' – acarreo3 = '0' (No se presta ya que el minuendo es mayor o igual al sustraendo, acarreo4 = '0')
5. '1' – '0' – acarreo4 = '1' (acarreo5 = '0')
6. '1' – '1' – acarreo5 = '0' (acarreo6 = '0')
7. '0' – '0' – acarreo6 = '0' (acarreo7 = '0')
8. '0' – '0' – acarreo7 = '0' (acarreo8 = '0')

Resultado = 00010011_2 o 19 en base decimal.

Uno de los detalles más interesantes se da cuando se restan dos números y el minuendo es menor que el sustraendo. Cuando esto ocurre, el número debe ser interpretado como negativo o complemento a 2. Por ejemplo, si se realiza la resta 72 – 94, el resultado es –22. Obviamente, para esto se han tenido que intercambiar las posiciones de los operandos, haciendo que 94 sea el minuendo y 72 el sustraendo.

En el caso de una resta binaria, el resultado de restar un minuendo menor que el sustraendo da el complemento a 2 del resultado de realizar la resta invirtiendo los roles del sustraendo y el minuendo. Por ejemplo, se sabe que 72 es 01001000_2 y 94 es 01011100_2. Al realizar la resta, se obtiene el siguiente resultado: 11101010_2, el cual es el complemento a 2 de 22.

Otra forma de realizar la misma operación resulta de sumar 72 con –94. Para el caso del sistema decimal, la operación sigue siendo una resta. Sin embargo, para el sistema binario se pueden aplicar las mismas reglas de la adición, sumando 72 con el complemento a 2 de 94. En este caso, –94 es 10100010_2. Entonces, sumando 01001000_2 + 10100010_2, se obtiene el valor 11101010_2, que resulta también ser –22 en complemento a 2.

La multiplicación binaria sigue un proceso muy similar al caso decimal. En esta última, los operandos son alineados uno encima del otro, y es el de menor longitud el de la posición inferior. Por ejemplo, si se desea multiplicar 123 x 14, el proceso para el cálculo será el siguiente:

$$\begin{array}{r} 123\ \times \\ 14 \\ \hline 492\ + \\ 123 \\ \hline 1722 \end{array}$$

Como se puede observar, el operando con la cifra más extensa se coloca en la parte superior y el operando de menor longitud en la parte inferior. Se multiplica el dígito menos significativo del operando inferior por cada uno de los dígitos de la cifra superior. Luego, se multiplica el segundo operando más significativo por todos los dígitos de la cifra superior, pero el resultado se coloca en la parte inferior, dejando una columna derecha libre. Finalmente, se suman los dígitos de cada columna respectiva y se obtiene el resultado.

En el caso binario el procedimiento es muy similar, con la ventaja de que los dígitos por multiplicar son bits y los resultados de cada multiplicación solo pueden ser '0' o '1'. La multiplicación de cada dígito binario es en realidad una operación AND. El procedimiento consiste en ubicar el operando de mayor longitud sobre el de menor longitud, de la misma manera que en el procedimiento de multiplicación de cifras decimales. Luego, se debe realizar la operación AND entre cada dígito del operando corto y cada dígito de la cifra superior, empezando desde el bit LSB hasta culminar con el bit MSB del operador inferior.

Por ejemplo, si se realiza la misma operación anterior en formato binario, se tiene que 123 = 01111011_2 y 14 = 00001110_2. Entonces, el procedimiento de multiplicación sería como se muestra a continuación:

				0	1	1	1	1	0	1	1_2 ×
				0	0	0	0	1	1	1	0_2
				0	0	0	0	0	0	0	0_2
			0	1	1	1	1	0	1	1_2	
		0	1	1	1	1	0	1	1_2		
	0	1	1	1	1	0	1	1_2			
0	1	1	0	1	0	1	1	1	0	1	0_2

Como resultado, se obtiene el valor binario 011010111010_2, que es equivalente a 1722. Cabe resaltar que este tipo de operación solo aplica para cifras positivas. Para multiplicar cifras en complemento a 2, se debe optar por otra metodología como sumas sucesivas.

1.1.5 Punto fijo y Coma flotante

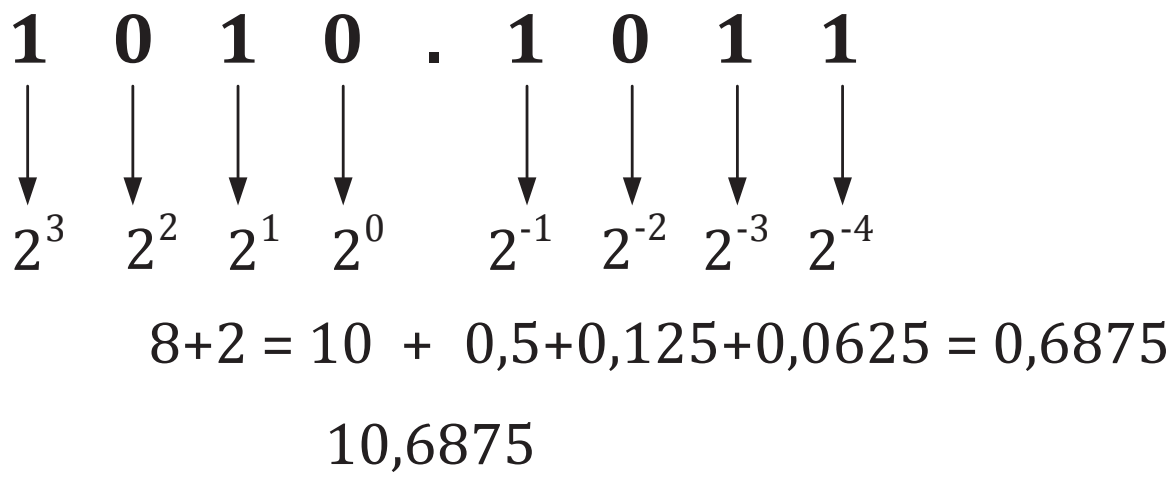
En muchas ocasiones es necesario representar una cifra de forma fraccional; para esto, existen dos maneras de hacerlo en código binario. Estos formatos son el Punto fijo y la Coma flotante.

El formato de Punto fijo es el más simple y no requiere de mucha carga computacional por parte del sistema embebido para ser operado. Este formato fue introducido al inicio de la década de 1980 y se encuentra basado en un tamaño fijo de bits, tanto para la parte fraccional como para la parte entera.

En el formato de Punto fijo, se debe establecer un punto decimal imaginario dentro del código binario. Por ejemplo, para una cifra binaria con el siguiente formato: 1010.1011_2, se muestra un byte que contiene un punto fijo entre ambos *nibbles*. Esta posición indica que los bits a la izquierda del punto decimal son considerados la parte entera de la cifra, mientras que los bits ubicados al lado derecho del punto son considerados la parte fraccionaria. En este caso, se han asignado 4 bits para la parte entera y 4 bits para la parte fraccionaria. Esto no necesariamente ha de ser así, ya que se pueden asignar diferentes tamaños para cada sección.

La interpretación de la cifra debe ser dada de dos maneras. Para la parte entera, se mantiene el formato entero. Para la parte fraccionaria, las potencias asignadas a la posición de cada bit presentan exponente negativo. En el gráfico 1.8 se muestra la forma como se obtiene el valor decimal de la cifra 1010.1011_2.

Gráfico 1.8. Conversión del formato punto fijo a decimal fraccionario



Elaboración propia.

Como se puede observar, la parte entera y fraccionaria se calcula por separado, para luego unirse en una sola cifra, incorporando el punto decimal. La mínima cantidad que puede ser expresada en formato Punto fijo es 0000.0000_2, que representa 0.0, mientras que el valor máximo es 1111.1111_2, que equivale a 15.9375. En muchos casos, el bit MSB de la parte entera es asumido como el signo de la cifra. En ese sentido, considerando el cuarto bit MSB de la parte entera el signo ('1' negativo y '0' positivo), el rango de valores máximo y mínimo que puede tener la cifra va desde –7.9375 hasta 7.9375 (tomar en cuenta que es diferente del formato complemento a 2, pues en este caso corresponde al formato complemento a 1[11]).

La precisión del formato de Punto fijo aumenta conforme el número de bits asignados a la etapa fraccionaria es incrementado. En la práctica, este método es utilizado por muchos sistemas embebidos de baja capacidad, aunque en muchas ocasiones es dejado de lado, por su falta de precisión.

Para mejorar la precisión de las cifras fraccionarias existe el formato de Coma flotante. En este formato se divide un conjunto de bits (normalmente entre 24 y 32 bits) en tres partes: signo, exponente y mantisa. En el gráfico 1.9 se muestra una representación del formato de Coma flotante para un número de 32 bits.

11 En el formato complemento a 1 el bit MSB representa el signo ('1' negativo y '0' positivo) y los bits restantes corresponden a la magnitud de la cifra.

Gráfico 1.9. Representación en Coma flotante para un número de 32 bits

S	EXP	MANTISA
S: Bit de signo (0=>+, 1=>–)		
EXP: Exponente (8bits)		
MANTISA: Base decimal (23 bits)		
Número = $(S)\times 2^{(EXP-BASE)}\times(1.0+\text{fracción}/(\text{base fracción}))$		

Elaboración propia.

Por lo general, el bit de signo (S) toma la posición más significativa de todo el registro de 32 bits y ocupa un espacio de 1 bit. El valor '0' indica que la cifra debe ser interpretada como positiva, mientras que el valor '1' representa una magnitud negativa.

El exponente especifica directamente la magnitud entera de la cifra en Coma flotante. El sector del exponente está compuesto por un número de bits, por lo general, menor que el asignado a la mantisa. El código binario del exponente representa un factor, al cual se le restará una cifra base. El resultado de esta resta será el exponente de un factor 2 al cual se le multiplicará un número fraccionario. La base viene a ser el máximo valor que se puede formar con el número de bits menos uno asignado al exponente. Esto implica que, para 8 bits de exponente, la base será 127, ya que es el máximo valor posible formado por 7 bits. Evidentemente, el exponente puede ser negativo, con lo cual se tendría la representación de una cifra netamente fraccionaria y menor que 1.

La mantisa especifica el valor fraccionario de la cifra en Coma flotante. Mientras más bits posea la mantisa, mayor será la precisión decimal de la cifra. Mientras que la base de la fracción es 2^n, donde n es el número de bits de la mantisa[12].

Como ejemplo, se cuenta con un número en Coma flotante compuesto por 13 bits. En este formato se utiliza un bit de signo, 4 bits para el exponente y 8 bits para la mantisa. El número presenta el siguiente código binario: 0100010100000_2. Entonces, la cifra decimal se obtendrá de la siguiente manera:

Bit MSB = '1' => Signo = –
Exponente = $1000_2 = 8$
Base = $2^{4-1}-1 = 7$
Fracción = $10100000_2 = 160$
Base de la fracción = $2^8 = 256$

Por lo tanto, la cifra se calcula de la siguiente manera:

$$\text{Cifra} = (-1)^1 2^{8-7} \times \left(1+\frac{160}{256}\right) = -3.25$$

En el mundo de los sistemas embebidos, el uso de números en formato Coma flotante es sumamente preferido por parte de los programadores para simplificar cálculos fraccionales y obtener una buena precisión. Sin embargo, la carga computacional que requiere este formato para realizar las ope-

12 Cfr. Pong Chu 2008.

raciones de suma, resta y multiplicación es bastante elevada para sistemas embebidos que no poseen una arquitectura optimizada para esto.

1.1.6 El sistema de numeración BCD

El formato BCD (*Binary Coded Decimal*) es bastante utilizado en los sistemas embebidos para el manejo de dispositivos periféricos, en los cuales se requiere presentar datos numéricos (por ejemplo, Displays 7 segmentos, módulos LCD alfanuméricos, entre otros). En este formato, cada dígito decimal es reemplazado por su equivalente en el sistema binario, pero solo utilizando cuatro bits. En el gráfico 1.10 se muestra el valor BCD para cada dígito decimal.

Gráfico 1.10. Códigos BCD para cada dígito decimal

Dígito decimal	Formato BCD
0	0000_2
1	0001_2
2	0010_2
3	0011_2
4	0100_2
5	0101_2
6	0110_2
7	0111_2
8	1000_2
9	1001_2

Elaboración propia.

Si la cifra que se representará en BCD es mayor que 9, se tendrán que agregar cuatro bits más al código para representar dos cifras decimales. En otras palabras, si la cifra decimal tiene *m* dígitos, esta misma cifra tendrá 4x*m* bits en formato BCD.

Por ejemplo, la cifra 9512 se codifica en BCD de la siguiente manera: 1001010100010010_2. Esto es diferente del formato binario, ya que la cifra 9512 requiere 14 bits para su representación.

Aunque parezca sorprendente, la aritmética en el formato BCD no resulta muy sencilla para un sistema embebido, por lo cual, normalmente se deben operar las cifras en formato binario, y luego aplicar una técnica para realizar la conversión de base binaria a formato BCD.

1.1.7 El sistema de numeración ASCII

El formato ASCII (*American Standard Code for Information Interchange*) es un sistema de codificación binario empleado para representar caracteres en el alfabeto latino. Como se trata de un estándar estadounidense, este formato representa todos los caracteres usados en el idioma inglés. El formato ASCII actualmente se utiliza para que los sistemas embebidos puedan configurar diversos dispositivos, tales

como pantallas LCD alfanuméricas, módems, cámaras digitales, etc. Inclusive, los lenguajes de programación incluyen dentro de su sintaxis la opción de interpretación de caracteres ASCII.

El estándar ASCII utiliza 7 bits para representar los caracteres del alfabeto latino. Existen otros caracteres extendidos que se obtienen al generar códigos de 8 bits y que permiten representar caracteres de los idiomas español o japonés.

Los primeros 32 caracteres del código ASCII son conocidos como «comandos de control». Estos cumplen con la función específica de configurar algunas opciones del texto sin imprimir ningún carácter en pantalla. Originalmente, el código ASCII fue usado para enviar información a dispositivos de texto, como teleimpresoras, pero ellos, al estar actualmente en obsolescencia, han dejado muchos de estos códigos sin una función útil. En el gráfico 1.11 se muestran los códigos ASCII para los caracteres de control. Los comandos que aparecen con un * son los que actualmente son utilizados en las diversas aplicaciones de los sistemas embebidos.

Gráfico 1.11. Códigos ASCII para los caracteres de control

Comando ASCII	Código hexadecimal	Función
NUL	0x00	Carácter nulo
SOH	0x01	Inicio de encabezado
STX	0x02	Inicio de texto
ETX	0x03	Fin de texto
EOT	0x04	Fin de transmisión
ENQ	0x05	Consulta
ACK	0x06	Acuse de recibo
BEL	0x07	Timbre
BS	0x08	Retroceso *
HT	0x09	Tabulación horizontal
LF	0x0A	Salto de línea *
VT	0x0B	Tabulación vertical
FF	0x0C	De avance
CR	0x0D	Retorno de carro *
SO	0x0E	Mayúsculas fuera
SI	0x0F	En mayúsculas
DEL	0x10	Enlace de datos/Escape
DC1	0x11	Dispositivo de control 1
DC2	0x12	Dispositivo de control 2
DC3	0x13	Dispositivo de control 3
DC4	0x14	Dispositivo de control 4
NAK	0x15	Confirmación negativa
SYN	0x16	Síncrono en espera
ETB	0x17	Fin de transmisión de bloque
CAN	0x18	Cancelar

Comando ASCII	Código hexadecimal	Función
EN	0x19	Finalización del medio
SUB	0x1A	Substítuto
ESC	0x1B	Escape *
FS	0x1C	Separador de fichero
GS	0x1D	Separador de grupo
RS	0x1E	Separador de registro
US	0x1F	Separador de unidad
DEL	0x7F	Eliminar *

Elaboración propia.

Posteriormente, quedan los caracteres ASCII imprimibles, los cuales son códigos que representan un símbolo alfanumérico. Estos caracteres están especificados entre los códigos del 33 al 126. En el gráfico 1.12 se muestra la lista de caracteres ASCII imprimibles.

Gráfico 1.12. Caracteres ASCII imprimibles

Código hexadecimal	Símbolo ASCII	Código hexadecimal	Símbolo ASCII	Código hexadecimal	Símbolo ASCII
0x20	Espacio ‘ ‘	0x40	@	0x60	`
0x21	!	0x41	A	0x61	a
0x22	“	0x42	B	0x62	b
0x23	#	0x43	C	0x63	c
0x24	$	0x44	D	0x64	d
0x25	%	0x45	E	0x65	e
0x26	&	0x46	F	0x66	f
0x27	‘	0x47	G	0x67	g
0x28	(	0x48	H	0x68	h
0x29	)	0x49	I	0x69	i
0x2A	*	0x4A	J	0x6A	j
0x2B	+	0x4B	K	0x6B	k
0x2C	,	0x4C	L	0x6C	l
0x2D	-	0x4D	M	0x6D	m
0x2E	.	0x4E	N	0x6E	n
0x2F	/	0x4F	O	0x6F	o
0x30	0	0x50	P	0x70	p
0x31	1	0x51	Q	0x71	q
0x32	2	0x52	R	0x72	r
0x33	3	0x53	S	0x73	s
0x34	4	0x54	T	0x74	t

Código hexadecimal	Símbolo ASCII	Código hexadecimal	Símbolo ASCII	Código hexadecimal	Símbolo ASCII
0x35	5	0x55	U	0x75	u
0x36	6	0x56	V	0x76	v
0x37	7	0x57	W	0x77	w
0x38	8	0x58	X	0x78	x
0x39	9	0x59	Y	0x79	y
0x3A	:	0x5A	Z	0x7A	z
0x3B	;	0x5B	[	0x7B	{
0x3C	<	0x5C	\	0x7C	|
0x3D	=	0x5D	]	0x7D	}
0x3E	>	0x5E	^	0x7E	~
0x3F	?	0x5F	_		

Elaboración propia.

1.2 Sistemas embebidos

El término «sistema embebido» hace referencia a todo circuito electrónico digital capaz de realizar operaciones de computación, generalmente en tiempo real, que sirven para cumplir una tarea específica en un producto.

Los sistemas embebidos no son equivalentes a los sistemas de cómputo usados en las *laptops* o en computadoras de escritorio que se venden en las tiendas tecnológicas, ya que los sistemas embebidos suelen tener recursos limitados y aplicaciones específicas que los hacen sumamente útiles en múltiples ambientes, como en el campo automotriz (sistemas de inyección de gasolina, alarmas contra robos, control de climatización, sistema de frenado ABS) o en teléfonos móviles, ipad, reproductores Blu-ray, refrigeradoras, alarmas de casas, lavadoras, cámaras fotográficas, instrumentación industrial, equipos médicos, Set Top Boxes, entre otros.

La arquitectura de un sistema embebido contiene un microprocesador dedicado capaz de ejecutar instrucciones a una determinada velocidad, controlada por una señal de reloj. De acuerdo con la arquitectura del microprocesador del sistema embebido, los recursos internos (periféricos) y la máxima frecuencia de operación, se define la potencia de procesamiento. Normalmente, este parámetro se mide en unidades de MIPS (millones de instrucciones por segundo).

El programa que ejecuta un sistema embebido es por lo general elaborado en lenguajes como el ensamblador, ANSI C, C++ o Basic. Algunos sistemas embebidos tienen la capacidad de ejecutar sistemas operativos limitados, conocidos como RTOS (Real Time Operating Systems), que permiten que el procesador ejecute diversas tareas, asignándoles una prioridad y un orden de ejecución. Algunos de estos sistemas operativos son el MQX de Freescale, µC/OS-II, FreeRTOS y µLinux, entre los más conocidos.

Una de las características actuales de todo sistema embebido, y muy importante, es el bajo consumo de energía. Como muchas de sus aplicaciones involucran dispositivos que utilizan baterías, los modos de bajo consumo de corriente (modo *sleep*) son sumamente importantes para que puedan ser utilizados en este tipo de aplicaciones.

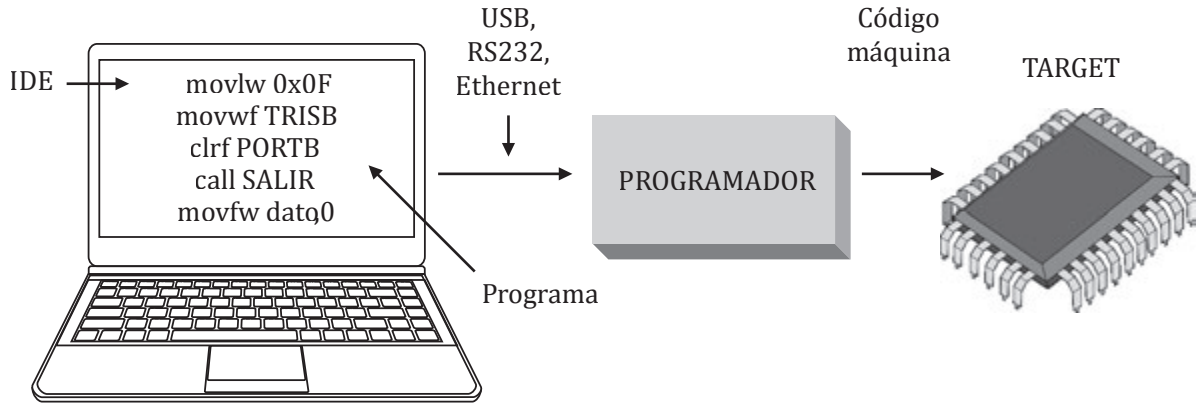
Un microcontrolador es un sistema embebido, y actualmente podemos encontrar microcontroladores en un solo chip, los cuales son sistemas computarizados completos, ya que contienen en su interior un microprocesador, unidades de memoria (de programa y datos), unidades de entrada-salida y periféricos. Hoy en día existen en el mercado microcontroladores de diversas arquitecturas (8 bits, 16 bits y 32 bits), diferentes marcas y modelos y dedicados para aplicaciones específicas.

1.3 Lenguajes de programación

El proceso de programación de un sistema embebido requiere inicialmente de un computador o *Host*, en el cual se ejecute una herramienta de *software* denominada IDE (*Integrated Development Enviroment*). Dentro de esta herramienta, se realiza la programación de instrucciones del sistema embebido, que, por lo general, puede incluir varios archivos fuente con partes del programa. La secuencia de instrucciones que especifica el programa en un archivo de texto que será grabado en el sistema embebido se conoce como *firmware*.

Cuando el programa es finalizado, se hace uso de un *software* llamado «compilador». Esta herramienta transforma el texto que contiene las instrucciones de forma secuencial y lo convierte en un código de máquina: un archivo compuesto de números binarios que va a ser enviado directamente a la memoria de programa del sistema embebido (denominado *Target*). En el gráfico 1.13 se muestra un esquema de la comunicación entre el *Host* y el *Target*.

Gráfico 1.13. Interfaz *Host-Target* para programación del sistema embebido



Elaboración propia.

Usualmente, se requiere una interfaz entre el *Host* y el *Target*, conocida como «el programador». Esta interfaz es un equipo que se puede conectar al *Host* a través del puerto RS232, USB, Ethernet o, antiguamente, en el puerto paralelo o LPT. Por otro lado, el programador se conecta al *Target* a través de un cable de conexión (cuyo estándar depende del fabricante del chip). El *software* IDE posee aplicativos que permiten enviar el archivo con el código máquina al programador. Este, a su vez, se comunica con una aplicación en el *Target* denominada «*intérprete*», que se encarga de grabar el código máquina en los registros de memoria de programa del sistema embebido.

Entre los lenguajes de programación que se pueden elegir para programar un sistema embebido, se tienen las siguientes opciones:

- Lenguaje ensamblador
- Lenguaje ANSI C
- Lenguaje C++
- Basic

El lenguaje ensamblador es el lenguaje más primitivo, conocido como «lenguaje de bajo nivel», ya que actúa a nivel de los registros internos del sistema embebido. Se caracteriza por poseer un conjunto de instrucciones específicas (normalmente aritméticas, lógicas, de manipulación de bit, de lectura y escritura en memoria y de salto) que permiten manipular los datos dentro del sistema embebido, siguiendo una secuencia especificada por el programador. Este lenguaje obliga al programador a conocer a fondo la arquitectura del sistema embebido, ya que las instrucciones suelen interactuar con los registros internos del procesador, con la unidad aritmética lógica, con las unidades de memoria y con los puertos de entrada y salida. Por ejemplo, en el lenguaje ensamblador se puede tener una secuencia de instrucciones como la siguiente:

1. *movwf var,0 ;Cargar el registro var con el valor en el registro W*
2. *movlw 0x56 ;Cargar el registro W con el valor literal 0x56*
3. *addwf var ,0,0 ;Sumar 0x56 (que se encuentra almacenado en el registro W) con ;el valor almacenado en var y guardar el contenido en W*

Los tres comandos presentados son una muestra del conjunto de instrucciones del lenguaje ensamblador del PIC18F[13] (el texto a la mano derecha del punto y coma representa un comentario y es ignorado por el compilador). Se puede notar que, para utilizar este lenguaje, es necesario saber que dentro de la arquitectura del procesador existe un registro W que es susceptible a la escritura y lectura de datos (instrucciones 1 y 2). También se puede observar que existe una variable de nombre *var*, que debe haber sido declarada en algún espacio de la memoria de datos (instrucciones 1 y 3). Por último, se puede identificar que el conjunto de instrucciones incluye una operación de suma aritmética (instrucción 3). Además, se debe saber que cada instrucción demora en ejecutarse 1 ciclo de instrucción[14], y que para llevar a cabo toda la operación se ha hecho uso de tres ciclos de instrucción, con lo cual se puede conocer el tiempo total de ejecución de las tres operaciones. En resumen, es claro que el lenguaje ensamblador obliga al programador a interactuar directamente con la arquitectura del sistema embebido.

El conjunto de instrucciones en lenguaje ensamblador varía de acuerdo con el fabricante del sistema embebido y con la arquitectura del procesador. Por ejemplo, Microchip posee una línea de microcontroladores de gama baja con un CPU[15] de código 16F87X. Dentro de esta línea se encuentran múltiples modelos de microcontroladores, como el PIC16F877A, PIC16F876 y PIC16F872, entre otros. Todos ellos comparten el mismo lenguaje ensamblador. Sin embargo, en la misma marca existe la familia de procesadores PIC18F4X, los cuales presentan un conjunto de instrucciones distinto, ya que la arquitectura del microprocesador es diferente.

En la actualidad, el lenguaje C es uno de los favoritos para la programación de sistemas embebidos. Fue creado en el año 1972 en los laboratorios Bell por Dennis Ritchie, con el objetivo de obtener

13 La sigla PIC18F hace referencia al modelo del procesador y las cifras o letras que continúan después de la «F» referencian al modelo específico del microcontrolador.

14 El ciclo de instrucción es el tiempo mínimo que demora una instrucción en ejecutarse.

15 Unidad central de proceso. Las siglas vienen del inglés *Central Processing Unit*.

un método para reescribir el sistema operativo Unix. Luego, en el año 1989, el lenguaje de programación C se convirtió al estándar del Instituto Nacional Estadounidense de Estándares (ANSI), conocido también como el lenguaje ANSI C. Este estándar fue utilizado para la programación de *software* en computadores y desde hace más de 14 años es utilizado en la programación de sistemas embebidos, con mucho éxito[16].

El lenguaje ANSI C

Se basa en ciertas reglas y operadores fundamentales que se aplican en la programación de cualquier sistema embebido. Esto permite que todos los programadores de sistemas embebidos, independientemente de su marca y modelo, logren hablar el mismo idioma de programación. Es así que, si se conoce este lenguaje, prácticamente se posee la capacidad de programar cualquier sistema embebido.

La rápida adopción del ANSI C como lenguaje de programación de sistemas embebidos se debe a los siguientes factores[17]:

- El ANSI C es un lenguaje universal y, por lo tanto, es muy fácil portar un programa diseñado para un tipo de sistema embebido a otro modelo o marca de sistema embebido.
- El lenguaje ANSI C facilita el trabajo en equipo.
- El ANSI C permite la creación de librerías específicas para el manejo de ciertas rutinas repetitivas. Esto facilita la programación, reduciendo el tiempo de desarrollo del *firmware*.
- El lenguaje C permite el encapsulado (rutinas de código que presentan argumentos de entrada y salidas específicas o retornos. Esto es conocido también como «funciones»).
- El ANSI C es un lenguaje con el que se puede implementar rápidamente algoritmos con fundamentos matemáticos complejos.
- Es posible introducir bloques de lenguaje ensamblador dentro del ANSI C.

Hace algunos años, los programadores de sistemas embebidos usaban el lenguaje ensamblador como único método para programar *firmware*. Cada instrucción en ensamblador o mnemónico tiene un equivalente en código binario que es la representación de la instrucción en la memoria de programa. Cuando se programa en ANSI C, el código debe ser transformado a ensamblador para que el compilador genere el código máquina. El proceso de conversión del ANSI C a ensamblador lo realiza un generador de código que se encarga de transformar el lenguaje ANSI C en un equivalente a ensamblador. Los mecanismos que utiliza el generador de código para transformar el ANSI C a ensamblador dependen de diversos factores: la arquitectura del sistema embebido, la calidad del generador de código y el uso de un generador gratuito o con costo de adquisición (en general, los gratuitos son menos eficientes).

Cuando se adoptó el ANSI C como lenguaje de programación para sistemas embebidos, este desplazó al ensamblador como el lenguaje de preferencia. Los programadores en ensamblador, en un principio, se mostraban escépticos hacia las ventajas aparentes del ANSI C. Hoy en día, a pesar del éxito del ANSI C como lenguaje de programación de sistemas embebidos, algunas desventajas prevalecen, lo cual hace que el ensamblador todavía resulte ser un lenguaje alternativo y necesario. Algunas desventajas de la programación en ANSI C para sistemas embebidos son las siguientes:

16 Cfr. Galeano 2009.

17 Cfr. Galeano 2009.

- Un programa en lenguaje ANSI C ocupa más espacio en la memoria de programa y requiere un mayor consumo de memoria de datos.
- Los buenos compiladores tienen un costo; los ensambladores siempre son gratuitos. Esto implica que contar con un generador de código que convierta el ANSI C a ensamblador tiene un precio. De acuerdo con la escala de pago, la eficiencia del ensamblador será mayor.
- Los compiladores son programas que pueden tener algunos inconvenientes y resultar ser no muy eficientes en algunas ocasiones.
- El lenguaje C no requiere que el programador posea un conocimiento profundo de la arquitectura del sistema embebido, lo cual puede llevar a cometer errores en la programación.

El lenguaje C++ es otra alternativa de programación de sistemas embebidos. Este lenguaje posee un nivel de abstracción más elevado que el ANSI C, y se basa en la programación orientada a objetos, en la cual un objeto es un arreglo de datos ordenados, de forma que facilita la programación. Muchos programadores en C++ se pueden adaptar rápidamente a la programación de sistemas embebidos haciendo uso de este lenguaje. Por ejemplo, la empresa Freescale provee un IDE de nombre CodeWarrior, que permite programar sus sistemas embebidos en lenguaje C++.

Otro lenguaje de programación, menos popular, es el Basic. Es un lenguaje de alto nivel, conocido por el mundo de los programadores que alguna vez desarrollaron aplicaciones con el *software* Visual Basic de Microsoft. De la misma manera, existen también compiladores que permiten programar sistemas embebidos en otros lenguajes, como el Pascal.

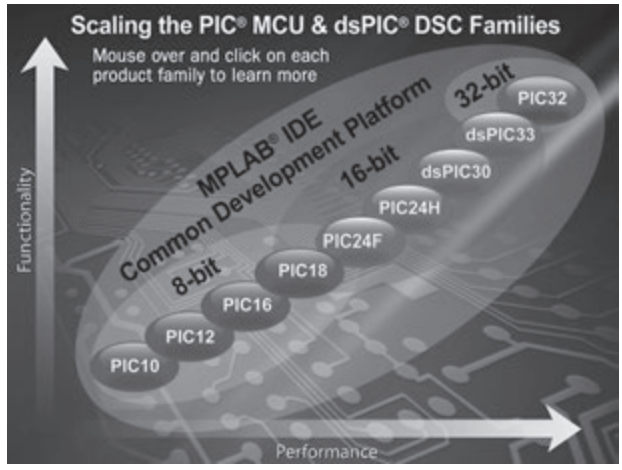
1.4 La familia PIC18F4X de la marca Microchip

Microchip Technology Inc. es una empresa estadounidense situada en el Estado de Arizona, y su rubro es la fabricación de microcontroladores, memorias y semiconductores analógicos utilizados para el diseño de equipos electrónicos en diversos campos de aplicación.

Dentro del rubro de microcontroladores que ofrece la compañía al mercado se encuentran tres grandes grupos generales: microcontroladores de 8, 16 y 32 bits de CPU. En el gráfico 1.14 se muestra la línea de productos de microcontroladores de Microchip ordenados de menor a mayor potencia de procesamiento. El número de bits especifica el tamaño del bus de datos que utiliza el procesador, y también la longitud de los registros internos de la memoria de datos.

La familia de 8 bits de CPU involucra cuatro arquitecturas: PIC10, PIC12, PIC16 y PIC18. Dentro de la familia PIC10, se encuentran microcontroladores de muy bajo costo y, por lo general, de 8 pines (en encapsulados del tipo DIP8 u 8DFN). En esta familia, los microcontroladores se caracterizan por ser de muy bajo consumo de energía y por operar a frecuencias inferiores a los 8 MHz. Poseen muy pocos módulos periféricos de entrada, generalmente módulos de comunicación serial síncrona y temporizadores. Los PIC10 contienen una memoria de programa y de datos bastante limitada en espacio. Esta familia de microcontroladores está diseñada para aplicaciones muy simples y que requieren del control de muy pocos periféricos, como el accionamiento de un motor, control de temperatura, receptores infrarrojos, etc.

Gráfico 1.14. Línea de microcontroladores de Microchip



Fuente: Microchip Technology Inc.

Por otro lado, la familia PIC12 posee una mayor capacidad en memoria (datos y programa) que su familia antecesora. Algunos modelos incluyen mayor velocidad de reloj y un módulo serial adicional. Su área de aplicación sigue siendo limitada.

En cuanto a la familia PIC16, fue introducida al mercado a mediados del año 1992. Este conjunto de microcontroladores presenta modelos mucho más variados, con concentraciones entre 14 y 64 pines en diversos empaques del encapsulado. La frecuencia máxima de operación logra alcanzar los 32 MHz, y en este tipo de familia ya se cuenta con diversos periféricos, tales como convertidores analógicos digitales, temporizadores, módulos de comunicación serial síncrona y asíncrona y generadores de PWM, entre otros que permiten realizar aplicaciones de una escala de complejidad mayor a sus antecesores.

La familia PIC18 apareció en el mercado mundial en el año 1999. Tomó como base la arquitectura del PIC16, manteniendo un procesador de 8 bits y compatibilidad en los registros de configuración, pero con mejoras gravitantes en la arquitectura: la velocidad máxima de operación ha sido incrementada hasta 64 MHz, algunos modelos incluyen módulos generadores de reloj (con un PLL[18] incorporado y un oscilador interno de 8 MHz), un conjunto de instrucciones más amplio (con 42 instrucciones adicionales), mayor capacidad de memoria de datos y mejoras en las funciones de los periféricos y periféricos adicionales (como USB y Ethernet), entre otras características. Adicionalmente, la arquitectura del PIC18 ha sido optimizada para la programación en lenguaje C. Todo esto permite que los PIC18 sean una de las mejores alternativas para las aplicaciones de mediano rango que incluyen desarrollos en el campo automotriz, biomedicina, domótica, sistemas de iluminación, control de motores, interfaces humanas (pantallas LCD gráficas, pantallas táctiles, pantallas alfanuméricas), en aplicaciones de radiofrecuencia y en sistemas de comunicación de redes de computadoras (Ethernet, USB, Bluetooth).

Posteriormente, Microchip lanzó al mercado otros procesadores de mayor envergadura para aplicaciones que requieren más capacidad de procesamiento de datos. En el año 2004 salieron al mercado los microcontroladores dsPIC30 y dsPIC33, que son sistemas embebidos de 16 bits de arquitec-

18 *Phase Locked Loop*. Circuito de lazo cerrado que permite elevar la frecuencia de una señal periódica.

tura, pero con el valor agregado de contener un pequeño bloque DSP[19] en su interior, el cual facilita la implementación de algoritmos de procesamiento digital de señales tales como Filtros digitales, Autocorrelación, Transformada rápida de Fourier (FFT), Filtros adaptivos, reconocedores de voz, etc. Los dsPIC fueron diseñados para cubrir aquellas aplicaciones que requieren de una mediana carga computacional y que necesitan de la ejecución de pequeños algoritmos de procesamiento digital de señales. Al año siguiente, Microchip produjo los microcontroladores de la familia PIC24 de 16 bits de CPU, para cubrir el área de mercado de aplicaciones que requieren de mayor velocidad y capacidad de procesamiento, para lo cual se utiliza un conjunto de instrucciones más complejo.

Finalmente, en el año 2007, Microchip fabricó la familia de microcontroladores PIC32MX, los cuales trabajan con un CPU de 32 bits y alcanzan una frecuencia máxima de operación de hasta 80 MHz. Esta línea de sistemas embebidos solo se encuentra disponible en empaques de montaje superficial con concentraciones de 64, 80 y 100 pines. Aquí la gama de aplicaciones es superior, lográndose obtener un dispositivo capaz de contener un número mayor de periféricos, en comparación con las familias antecesoras, ya que posee opciones de manejo de bajo consumo, módulo DMA[20], nuevos periféricos, como módulos USB OTG[21] y controladores Ethernet mejorados.

1.5 Herramientas de desarrollo para la marca Microchip

Para poder comenzar a realizar desarrollos electrónicos con las herramientas de Microchip es necesario tener en cuenta que se requieren tres tipos de herramientas de desarrollo:

- Herramienta IDE o *software* de desarrollo
- Programador/Depurador
- El sistema embebido o *Target*

Dentro de las herramientas de *software* que se encuentran disponibles para la elaboración de código se pueden encontrar diversas opciones en internet. Algunas son gratuitas y otras presentan un costo de adquisición. El IDE más popular para programar los sistemas embebidos de Microchip no puede ser otro más que el *software* ofrecido por el fabricante: el MPLAB X IDE.

El MPLAB X IDE es un programa que puede descargarse gratuitamente desde la página web de Microchip y que se instala sobre cualquier sistema operativo Windows, Mac OS o Linux sobre una plataforma de 32 o 64 bits. El *software* incluye un ambiente de desarrollo gráfico, con un editor de texto para introducir los programas, un compilador de lenguaje ensamblador, un depurador que permite verificar los errores del programa y un enlazador que transforma el código de programa en el lenguaje de máquina que posteriormente será grabado en la memoria no volátil del sistema embebido. El MPLAB X IDE posibilita la programación de todas las familias de microcontroladores de Microchip, empezando desde la arquitectura más limitada (como la PIC10) hasta la más avanzada (PIC32MX).

19 *Digital Signal Processor*.

20 *Direct Memory Access*. Modos de acceso a memoria sin ocupar recursos del CPU.

21 *USB On The Go*. Tecnología que permite a un dispositivo embebido funcionar como maestro o *Host* USB.

Adicionalmente, el MPLAB X IDE incluye un simulador de *firmware* que es de gran ayuda para el proceso de depuración de las instrucciones paso a paso. Esta herramienta permite visualizar el resultado de cada instrucción, mostrando los contenidos de cada registro en la memoria de datos. El simulador opera tanto para el lenguaje ensamblador como para el lenguaje ANSI C.

Si se requiere la programación del microcontrolador en lenguaje ANSI C, es necesario descargar el *software* MPLAB XC8, herramienta que se descarga de forma independiente de la página web del fabricante y que posibilita la programación en lenguaje C de la familia PIC18. Esta herramienta incluye librerías para el manejo de periféricos como LCD, SPI, I^2C, PWM y USART, entre otros. Además, contiene librerías matemáticas y de manipulación de memoria. El compilador XC8 posee tres modos de operación: gratuito (*FREE*), *Standart* y *PRO*. El último modo posee un nivel de optimización 50% más eficiente que el modo *FREE* y de 20 a 25% que el modo *Standart.* La descarga gratuita incluye una versión de evaluación (*FREE*) que opera en modo PRO durante los primeros 60 días después de su instalación. Pasado este plazo, el *software* ingresa en un modo de baja optimización de código. Si se desea tener acceso a la licencia del MPLAB XC8 de manera permanente, se puede adquirir del fabricante.

Existen en el mercado otras herramientas, como el CCS IDE Compiler, que es un *software* que permite programar microcontroladores de Microchip de la familia PIC10, PIC12, PIC14, PIC16, PIC18, PIC24 y dsPIC. Esta herramienta contiene un conjunto de librerías para manejar con mucha facilidad los microcontroladores haciendo uso del lenguaje ANSI C. El *software* se puede descargar gratuitamente de internet a través de la página web (www.ccsinfo.com). Pero, al igual que el MPLAB XC8, posee un tiempo de expiración y también presenta un costo para una licencia permanente. Adicionalmente, se pueden encontrar otras herramientas en el mercado, como MikroC o Mikro Basic de la empresa Mikroelektronica, cada una con una versión gratuita y una licencia con costo específico. Existen, por supuesto, muchas otras herramientas de programación para microcontroladores Microchip, pero se han expuesto las más populares.

En cuanto a los programadores, en el mercado se cuenta con diversos modelos que son compatibles con el MPLAB X IDE. Esto permite que, una vez conectados al computador, puedan descargar la aplicación directamente al *Target* sin necesidad de otro *software* adicional. El MPLAB X IDE es compatible con los siguientes programadores:

1. MPLAB ICD 3. Es un programador de alta velocidad que permite la grabación de microcontroladores de 8 y 16 bits de CPU y las familias dsPIC y PIC32MX. Se conecta al computador a través del puerto USB y al *Target* a través de un conector RJ11. Posibilita la depuración de los programas en tiempo real (simulación en *hardware*) y puede proveer la alimentación del circuito (5 V) aprovechando la conexión USB. No incluye nada más que un cable USB enrollable y el conector RJ11. Posee la ventaja de que puede programar al microcontrolador dentro de su circuito de desarrollo, impidiendo la extracción del mismo y permitiendo la programación de sistemas embebidos en empaque superficial (que normalmente no pueden ser extraídos del circuito por encontrase soldados en la tarjeta impresa). En el gráfico 1.15 se muestra una imagen del programador.

Gráfico 1.15. El programador MPLAB ICD 3

Fuente: Microchip Technology Inc.

2. El PICkit 3. Es un programador de bajo costo en forma de llavero. Cuenta con un conector hembra de seis pines que se puede adaptar a cualquier circuito para realizar la programación del microcontrolador. No requiere alimentación externa y se conecta al computador a través del puerto USB. Este grabador permite la programación de todas las familias de microcontroladores de Microchip, pero no es posible la depuración de las familias PIC32MX avanzadas. Posee la ventaja de que puede programar microcontroladores con voltajes de alimentación entre 2 a 6 V.

El PICkit 3 se vende junto con un cable USB y un CD que incluye la última versión del MPLAB X IDE. En el gráfico 1.16 se puede ver una imagen de este programador.

Gráfico 1.16. El programador PICkit 3

Fuente: Microchip Technology Inc.

3. El PICkit 2. Este grabador es el antecesor del PICkit 3. Permite la grabación de casi todas las familias de microcontroladores de Microchip, a excepción de los PIC32MX7, PIC32MX6 y PIC32MX5. Tampoco permite la depuración de ninguno de los tipos restantes de la familia PIC32. Posee las mismas dimensiones físicas y conector que el PICkit 3, pero con la diferencia de tener un empaque plástico de color oscuro. También se conecta al computador a través de un cable USB. En el gráfico 1.17 se muestra una imagen de este programador.
4. MPLAB PM3. Es un programador universal de fácil manejo que permite programar los microcontroladores conectados al computador o en modo *Stand alone* (sin conexión al computador). Puede cargar los archivos de programación desde una memoria SD que se conecta al equipo. Se conecta al computador a través del puerto RS232 o USB y soporta la gran mayoría de microcontroladores PIC y dsPIC.

Incluye los cables RS232, USB, la fuente de voltaje, el conector ICSD para la conexión al *Target* y un CD con la última versión del MPLAB X IDE. En el gráfico 1.18 se muestra una imagen del programador.

Gráfico 1.17. El programador PICkit 2

Fuente: Microchip Technology Inc.

Gráfico 1.18. El programador MPLAB PM3

Fuente: Microchip Technology Inc.

5. MPLAB REAL ICE. Es un programador de última generación de muy alta velocidad, que permite programar los microcontroladores de 8 y 16 bits y los dsPIC de Microchip. Se conecta al computador vía el puerto USB 2.0 y al *Target* a través de un conector RJ11. Permite la conexión de un cable de hasta 3 metros de longitud. En el gráfico 1.19 se puede ver una imagen de este programador.

Gráfico 1.19. El programador MPLAB REAL ICE

Fuente: Microchip Technology Inc.

1.6 Características específicas del PIC18F4550

Se ha elegido el PIC18F4550 como modelo para todos los ejemplos de este libro debido a que es uno de los microcontroladores más populares de toda la familia PIC18. Además, posee un empaque tipo DIP40 que lo hace fácilmente manejable y es de un costo accesible en el mercado nacional. Este modelo de microcontrolador contiene casi todos los periféricos que se pueden encontrar en la familia PIC18, con el valor agregado de contar con un módulo USB interno capaz de operar en los modos *Low Speed* y *Full Speed*.

El PIC18F4550 puede conectarse a un computador a través del puerto USB bajo dos velocidades diferentes: *Low Speed* y *Full Speed*. La primera trabaja a una velocidad de 1.5 Mbps, y la segunda, a 12 Mbps.

El módulo USB obliga al PIC18F4550 a tener un nuevo esquema de control de reloj, en el cual se puede producir una frecuencia de 48 MHz a partir de un cristal oscilador base de 4 MHz haciendo uso de un PLL interno. A partir de la señal más alta, se configura un *Postscaler* y un *Prescaler*, que permiten dividir esta frecuencia por diversos factores y configurar la frecuencia de reloj del microprocesador. La señal de 48 MHz alimenta directamente al módulo USB y es necesaria para que este pueda operar en el modo *Full Speed* (12 Mbps).

El microcontrolador PIC18F4550 incluye tres tipos de memoria: memoria de programa, memoria de datos y memoria EEPROM. La memoria de programa tiene una capacidad total de 32 kB, pero solo se aprovecha la mitad del espacio, debido a circunstancias que se explicarán en el siguiente capítulo. El ancho de la memoria de programa es de 16 bits. La memoria de datos es de una capacidad de 2 kB y contiene registros de 8 bits de longitud. La memoria EEPROM permite grabar información no volátil de 256 bytes de capacidad. Además, el PIC18F4550 contiene una pila de 31 registros de 21 bits para el almacenamiento de las direcciones de la memoria de programa durante las instrucciones de salto y las interrupciones.

El PIC18F4550 cuenta con 20 fuentes de interrupción y 35 pines de entrada y salida. Adicionalmente, posee 13 canales de entrada analógica, un módulo I^2C, otro SPI y un USART mejorado. Cuenta, además, con dos módulos generadores de PWM, que también pueden actuar como entrada de captura y comparación de salida. Además, tiene dos comparadores analógicos, un módulo de cambio de estado de cuatro pines y tres entradas de interrupción externas.

Por otro lado, el PIC18F4550 puede operar bajo dos tensiones de alimentación: 5 V y 3.3 V, y presenta dos modos de bajo consumo para el ahorro de energía. Los modos de bajo consumo inhabilitan múltiples funciones del microcontrolador, incluyendo el CPU, pero pueden ser despertados bajo ciertos eventos externos relacionados con algunos periféricos, como las interrupciones externas o los comparadores analógicos.

Durante los siguientes capítulos se propondrán diseños basados en el PIC18F4550 que pueden ser fácilmente implementados en un *hardware* basado en un circuito impreso, como en un Protoboard, para la verificación del funcionamiento. También es posible utilizar un *software* de simulación, como Proteus ISIS, para la validación de los programas de ejemplo de este libro.

Preguntas de repaso

1. Exprese el número en base decimal 9999 en código binario y en código hexadecimal.
2. Convierta la cifra 1111000010100011_2 a base decimal y represéntela en base hexadecimal.
3. ¿Cuántos *nibbles* posee y cuántos bytes alberga el número 0x98756421 en base hexadecimal?
4. Suponga que la variable A = 11001100_2. ¿Cuál es el resultado de la negación de esta variable?
5. Obtenga el resultado en base hexadecimal de la siguiente operación OR: 0xFA32 OR 0001001100101010_2.
6. Suponga que A = 0xF241 y B = 3450. Obtenga el resultado de la operación A XOR B en base binaria.
7. Realice la siguiente operación: $C = \overline{A \oplus B} + \overline{A} + B$, si A = 0x9021 y B = 0xAF00.
8. Obtenga la sumatoria de las siguientes cifras en base binaria:
   a. $10010001_2 + 110000010_2$
   b. $0x34A8 + 1111110011001010_2$
   c. $154 + 0x91$
   d. $10101010_2 + 01010101_2$
9. Presente las siguientes cifras en su código binario complemento a 2:
   a. 56
   b. 130
   c. 300
   d. 1
10. Obtenga las siguientes restas de cifras en base binaria:
    a. $10000000_2 - 01111111_2$
    b. $10000000_2 - 100000001_2$
    c. $111111111_2 - 00011001_2$
11. Multiplique las siguientes cifras y obtenga el resultado en base binaria:
    a. $00000011_2 \times 00000100_2$
    b. $00001111_2 \times 00001111_2$
    c. $10000000_2 \times 00001000_2$
12. Se tiene una variable en formato de Coma flotante de 13 bits. Tiene asignados un bit de signo, 5 bits de exponente y 7 bits de mantisa. El código binario que contiene la variable es el siguiente: 1001010100101_2. ¿Cuál es la cifra decimal que representa?
13. Represente las siguientes cifras decimales en formato de Punto fijo:
    a. 45.5
    b. –12.25
    c. 100.375
    d. 100.125
14. Si una variable contiene la representación en código ASCII de la letra 'A' y se adiciona el código binario 00001001_2, ¿cuál será la nueva cifra codificada en ASCII que albergará esta variable?

15. Convierta las siguientes cifras a su formato equivalente en BCD:
    a. 2012
    b. 0xA9
    c. 10000000_2
16. ¿Cuáles son las características más relevantes de un sistema embebido?
17. ¿Cuál es la diferencia entre *hardware*, *software* y *firmware*?
18. ¿Qué diferencia existe entre un compilador y un enlazador?
19. Enumere los lenguajes de programación utilizados para la programación de sistemas embebidos.
20. Explique cuáles son las diferencias más relevantes entre un microcontrolador de la familia PIC18F4 y un dSPIC30F.

Respuestas

Respuestas del capítulo 1:

http://bit.ly/1K5Ul3q

Capítulo 2. Arquitectura del microcontrolador PIC18F

Para comenzar a detallar las partes principales del microcontrolador PIC18F, es necesario definir ciertos conceptos generales sobre la arquitectura de cualquier sistema embebido en general. En este capítulo se describen conceptos básicos sobre los microcontroladores y luego se ahonda en las características específicas de la arquitectura del PIC18F.

Antes de revisar las partes del microcontrolador PIC18F, se deberá tener en claro el concepto de computador. Un computador es un sistema complejo formado por dos elementos fundamentales: *hardware* y *firmware*. La parte física del computador se compone de los siguientes elementos principales:

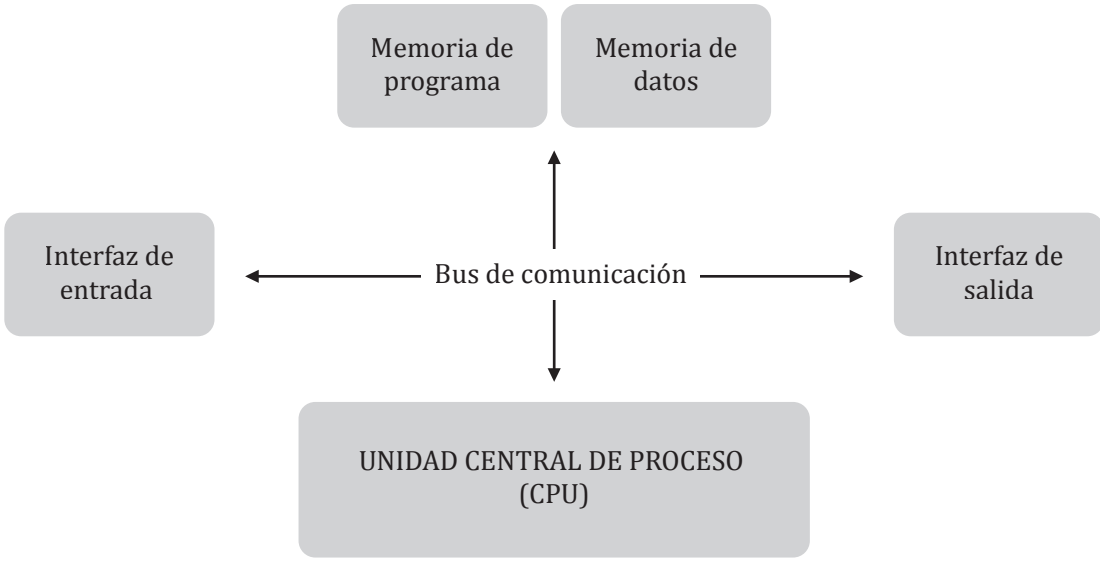
1. **Procesador**. Es el elemento encargado de ejecutar todas las instrucciones y de configurar los recursos del sistema. Un computador limitado, como un sistema embebido, usualmente posee un solo procesador, pero existen computadoras avanzadas que tienen múltiples procesadores que realizan tareas y cálculos muy complejos.
2. **Dispositivos de entrada**. Toda computadora está diseñada para relacionarse con el mundo exterior. Esto implica que los programas ejecutados pueden interactuar con datos que provienen de un ambiente externo al procesador. Entre los dispositivos de entrada más conocidos se tienen teclados matriciales, ratones, teclados PS/2, lectores de códigos de barras, lectores RFID, sensores, etc.
3. **Dispositivos de salida**. Un computador debe interactuar con el usuario y, para esto, debe presentar la información que procesa de manera que pueda ser visualizada e interpretada por un ser humano. Existen en el mercado una serie de dispositivos para este propósito, tales como monitores LCD, Displays 7 segmentos, módulos LCD alfanuméricos, impresoras térmicas, diodos emisores de luz (LED), etc.
4. **Unidades de memoria**. Las instrucciones que el procesador ejecuta y los datos que procesa deben estar almacenados en algún tipo de memoria, de manera que el procesador pueda escribir y leer directamente sobre este dispositivo. La memoria se puede dividir en dos tipos principales: memoria de instrucciones y memoria de datos. La primera tiende a ser de tipo no volátil; es decir, es una memoria que no pierde su contenido a pesar de prescindir de la energía que lo alimenta. Mientras que el segundo tipo de memoria es volátil, lo cual implica que, al sustraer la energía, la información almacenada se perderá.

Históricamente, la computadora digital, tal como se conoce hoy en día, es el resultado de una serie de experimentos realizados en la época de la Segunda Guerra Mundial (1939-1945). A partir de aquellos tiempos se empezaron a implementar múltiples mejoras que poco a poco fueron dando como resultado computadoras capaces de hacer cálculos y operaciones cada vez menos limitadas. En aque-

lla época, el matemático húngaro John von Neumann, que trabajaba para el programa de armamento nuclear de los Estados Unidos, comenzó el diseño de lo que sería el primer modelo de un computador.

El principal aporte de Von Neumann y su equipo de ingenieros fue la invención de una computadora cuya unidad de memoria de programa y de datos se encontraba compartiendo el mismo bus de datos y direcciones. Esto implicaba que el acceso desde el procesador a una instrucción o a un dato no podía darse de forma simultánea. El gráfico 2.1 muestra un esquema de la propuesta de diseño del computador de Von Neumann, donde se puede ver que un solo bus comunica al procesador con el resto de elementos del computador. Esto genera un cuello de botella entre el acceso a las unidades de memoria y las interfaces de entrada y salida[22].

Gráfico 2.1. Arquitectura del computador Von Neumann[23]



Adaptado de http://minnie.tuhs.org/CompArch/Lectures/week01.html

Por otro lado, una de las ventajas de la arquitectura de Von Neumann es su simpleza, lo cual hace que su implementación sea sencilla y de bajo costo.

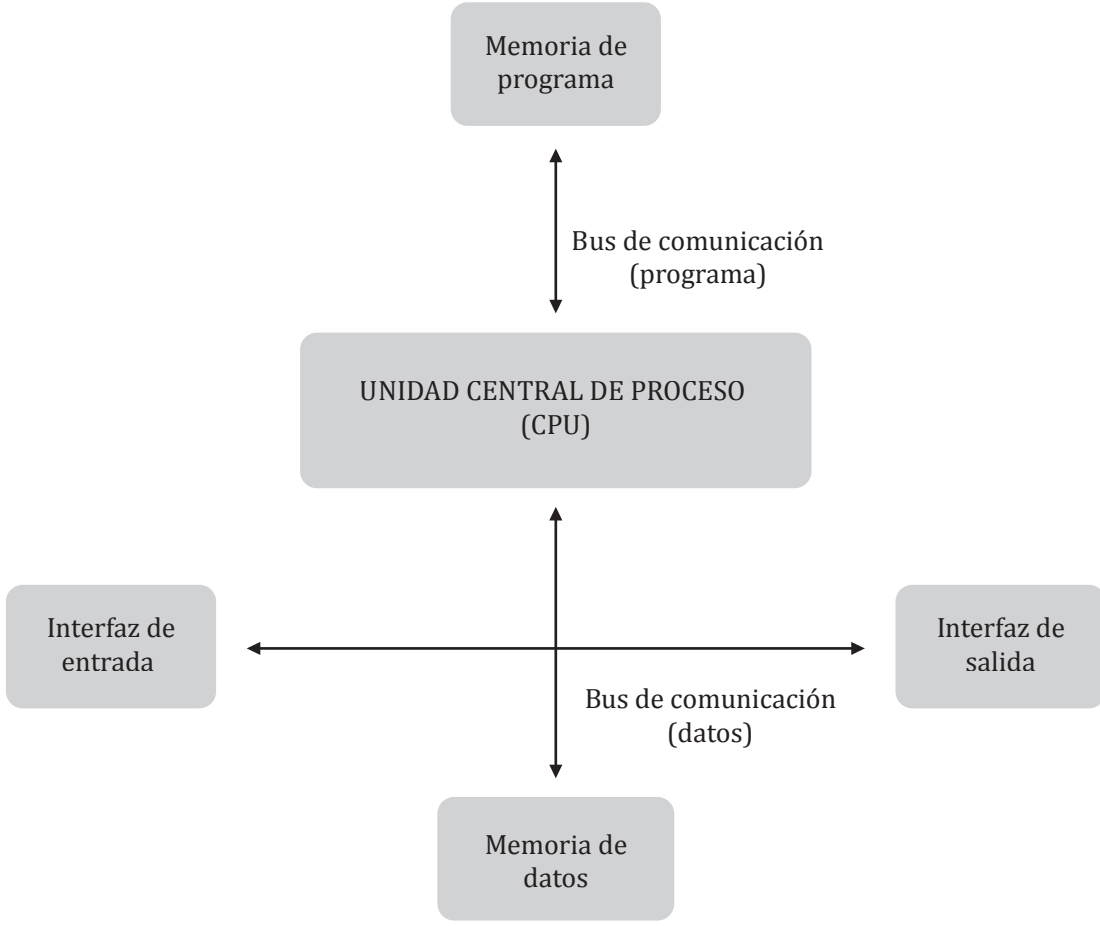
Diez años después de la propuesta de Von Neumann, se realizaron otros proyectos de diseño en la Universidad de Harvard, en los cuales se desarrollaron e implementaron una serie de computadoras conocidas como Mark 1, Mark 2, Mark 3 y Mark 4, las cuales modificaron el diseño inicial de Von Neumann en la etapa de acceso a la memoria, permitiendo que exista un camino independiente entre el procesador con la memoria de datos y con la memoria de programa. Esta modificación resultó ser más eficiente, ya que el acceso a las instrucciones se podía realizar de manera paralela con el acceso a los datos. Sin embargo, para la década de 1950, la tecnología no era lo suficientemente avanzada como para soportar esta arquitectura, por lo cual el diseño resultó muy costoso y no se volvió muy

22 Cfr. Sánchez y Canton 2007.

23 También llamado «Arquitectura del computador Princeton».

popular hasta varios años después, cuando la tecnología se hizo accesible. En el gráfico 2.2 se muestra el modelo de la arquitectura Harvard[24].

Gráfico 2.2 Arquitectura del computador Harvard



Adaptado de Sanchez y Canton 2007: 142.

2.1 La Unidad central de proceso (CPU)

La Unidad central de proceso, o simplemente el procesador, se compone de las siguientes partes:

- **Registros**. Los registros son pequeñas unidades de memoria incorporadas dentro del procesador. Cumplen con la función específica de almacenar información útil para la ejecución de las instrucciones y permiten su rápido acceso.
- **Unidad aritmética lógica (ALU)**. Es la parte del procesador encargada de ejecutar todos los pro-

24 Cfr. Ibrahim 2008.

cesos aritméticos y lógicos. La ALU recibe información de los registros internos o de una memoria. Esta información actúa como los operandos de la ALU, la cual ejecuta las operaciones que modifican estos valores, generando un resultado que puede tener como destino los registros del CPU o una dirección de memoria. Entre las operaciones aritméticas típicas de una ALU se encuentran la suma y la resta, mientras que entre sus operaciones lógicas básicas se tienen las operaciones AND, OR, NOT y XOR.

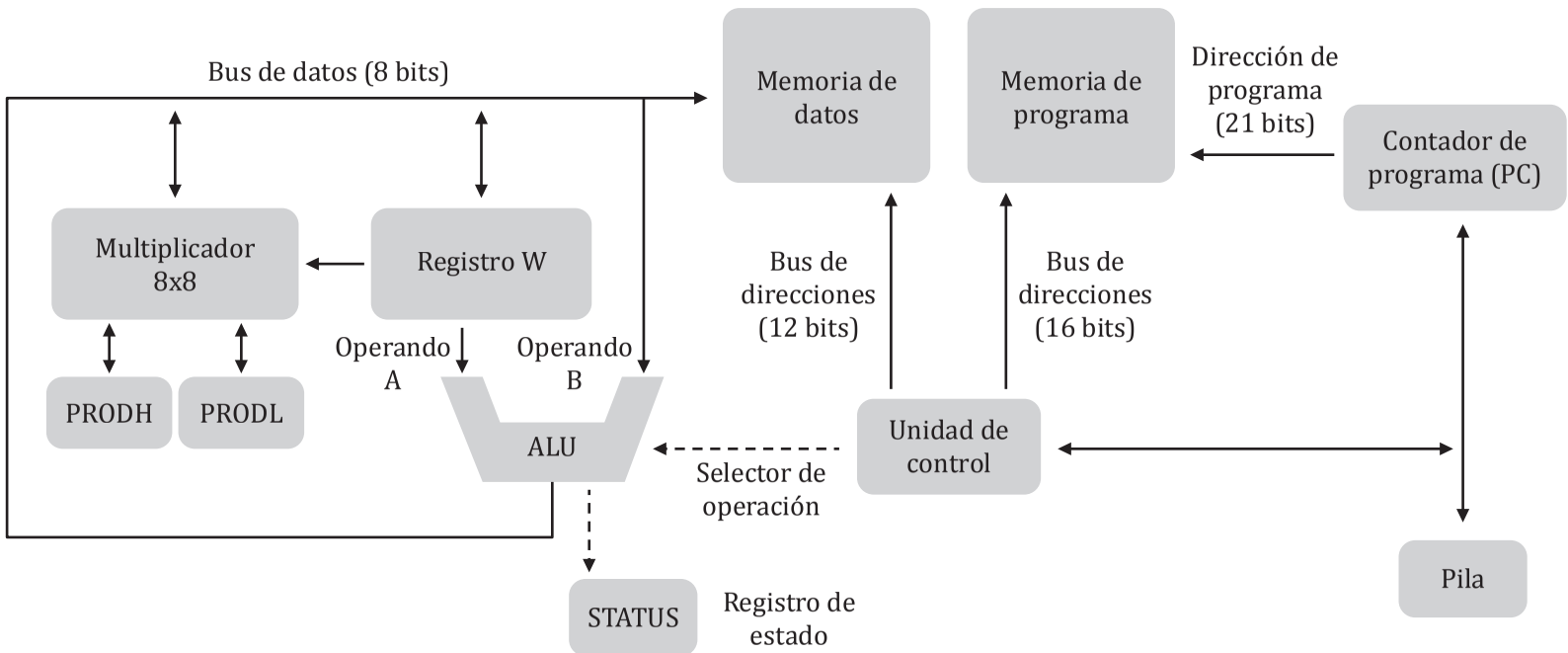
- **Unidad de control**. Esta etapa es la encargada de extraer las instrucciones de la memoria de programa, decodificarlas y ejecutarlas. Esta unidad trabaja de forma cercana con el Contador de programa (PC), que es un registro interno del procesador encargado de apuntar a la siguiente dirección de memoria de programa que la Unidad de control deberá ejecutar.
- **Reloj del sistema**. Un procesador es un sistema secuencial. Esto significa que solo una instrucción se ejecuta a la vez después de un tiempo Δt. Mientras más pequeño sea Δt mayor será el número de instrucciones por segundo que podrá ejecutar el procesador (aumentará el MIPS). Para que se pueda fijar un Δt de ejecución de instrucciones, todo procesador utiliza una señal de reloj externa cuya frecuencia (F_{RELOJ}) determina el Δt, donde $\Delta t = \frac{1}{F_{RELOJ}} \times N$, en que N es el número de periodos de reloj que la Unidad de control requiere para ejecutar una instrucción completa. La frecuencia de reloj se mide en unidades de Hertz (Hz), lo cual representa la cantidad de ciclos por segundo que el periodo de la onda de reloj se repite.

Los microcontroladores PIC18F obtienen en la mayoría de aplicaciones su señal de reloj de un cristal de cuarzo que se conecta de forma externa entre los pines OSC1 y OSC2. El valor de frecuencia nominal de este cristal determina la velocidad de procesamiento. Por ejemplo, una frecuencia de reloj típica utilizada en los microcontroladores PIC18F es la de 4 MHz o 4 millones de ciclos por segundo. Todo procesador maneja un rango de frecuencias de reloj máximo y mínimo de operación. Mientras mayor sea la frecuencia de reloj usada, más rápido será el procesamiento o ejecución de las instrucciones. Pero el aumento de frecuencia trae un costo: el consumo de energía aumentará por parte del sistema embebido.

Los procesadores también se caracterizan por el número de bits que pueden manipular en una sola operación. Cuando se hace referencia a un procesador de 8 bits, 16 bits o 32 bits, se está describiendo la longitud del bus de datos del microprocesador. La familia de procesadores PIC18F4X, que será tratada en este libro, trabaja con 8 bits de longitud.

En el gráfico 2.3 se puede observar un diagrama de bloques de la arquitectura del procesador PIC18F4. Aquí se pueden visualizar los registros internos del procesador, la ALU y la Unidad de control. Las unidades de memoria serán tratadas en el siguiente subcapítulo. A continuación se explica en detalle cada parte del esquema de la arquitectura del procesador PIC18F4X.

Gráfico 2.3. Diagrama de bloques de la arquitectura del procesador PIC18F4X



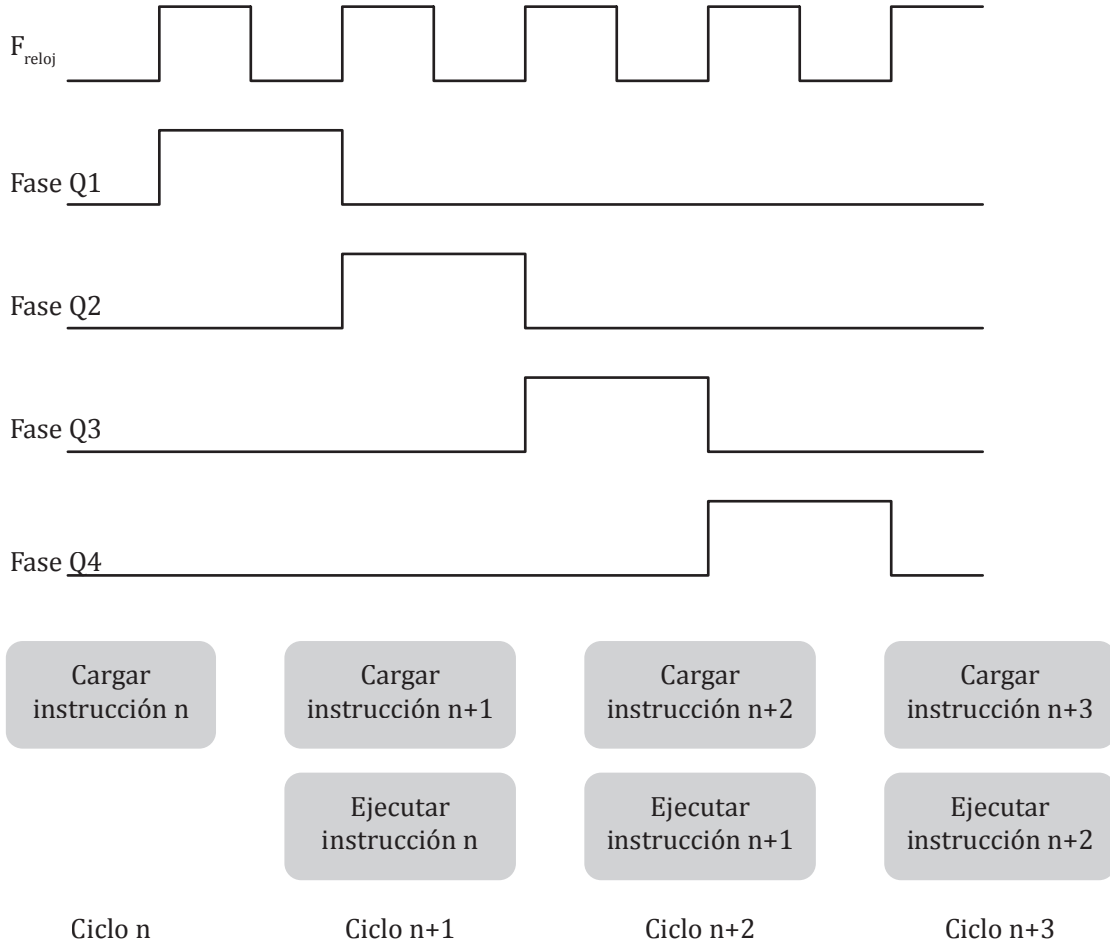
Fuente: Hoja de datos del PIC18F4550.

Las unidades del procesador PIC18F4X son las siguientes:

- **Contador de programa (PC)**. Es un registro de 21 bits capaz de direccionar hasta 2^{20} posiciones de memoria o instrucciones, ya que el bit LSB del PC siempre es '0'. El código binario contenido en este registro hace referencia a una posición en la memoria de programa en donde se encontrará la siguiente instrucción a ser ejecutada por la Unidad de control. Obviamente, las 2^{20} direcciones no son utilizadas en su totalidad, ya que las capacidades de las memorias de programa en los microcontroladores PIC18F no superan los 64 kB. El PC se autoincrementa en dos unidades normalmente. Además, el PC se conecta a la Unidad de control y ejecución de instrucciones a través de un bus independiente. La ALU puede alterar el contenido del PC ejecutando ciertas operaciones que pueden cambiar el orden de su contenido. Existen también, otras instrucciones, como *goto* y *call*, que son capaces de modificar el contenido del PC de forma directa.
- **Pila**. Está formada por 31 registros de 21 bits que se encuentran conectados al PC. La función principal de la Pila es guardar valores antiguos del PC, de manera que permita otorgar la capacidad de memorizar cuáles fueron las direcciones anteriores que cambiaron el flujo del programa. Normalmente, las instrucciones de salto que requieren un retorno futuro como *call* o *rcall* tienden a cargar la pila con el valor del PC antes del salto. La pila se direcciona a través de un registro denominado STKPTR (*Stack Pointer* o Puntero de pila) que solo contiene 5 bits de longitud. En el siguiente subcapítulo se presentan más detalles sobre las unidades de memoria, lo cual incluye a la Pila.
- **Pipeline**. Forma parte de la Unidad de control y está conformado por dos registros de 16 bits. Un registro contiene el código binario de la instrucción que ha sido leída de la memoria de programa en la ubicación apuntada por el PC. El otro registro alimenta al circuito decodificador y su contenido es la instrucción que va a ser ejecutada. Este esquema permite que el proceso de ejecución de instrucciones pueda leer una nueva instrucción desde la memoria de programa mientras la instrucción anterior está siendo ejecutada. En el gráfico 2.4 se muestra un esquema de este proceso de lectura y ejecución de instrucciones. Como se puede ver, ambos procesos en simultáneo requieren de cuatro ciclos de reloj, por lo cual el mínimo tiempo de ejecución de una instrucción es $4 \times T_{RELOJ}$, donde $T_{RELOJ} = \frac{1}{F_{RELOJ}}$. Este tiempo también se conoce como «ciclo de instrucción». Si la instrucción a ejecutar es de salto (*goto*, *call*, *rcall* u otra), el proceso requiere otro esquema de ejecución, por lo cual se agrega un ciclo más de ejecución al Pipeline (en total, demoraría dos ciclos de instrucción).
- **Decodificador de instrucciones**. También forma parte de la Unidad de control y consiste de una circuitería encargada de decodificar cada campo del registro de 16 bits de instrucciones del Pipeline y generar las direcciones y datos que van a alimentar a la Unidad de ejecución de instrucciones. Este proceso toma solamente dos ciclos del Pipeline para ejecutar dos tareas. En la fase Q1 se autoincrementa el PC y su contenido se carga al bus de direcciones de la memoria de programa. En la fase Q4 se carga el contenido de la dirección apuntada por el PC en el registro de instrucciones y, al mismo tiempo, la instrucción que antes estaba en este registro pasa al registro de entrada de la Unidad de ejecución de instrucciones.
- **Unidad de ejecución de instrucciones**. Esta unidad toma como entrada el código binario de 16 bits con la instrucción entregada por el decodificador de instrucciones. El proceso de ejecución normalmente implica leer un registro de la memoria de datos, colocar los operandos en la ALU, seleccionar la operación, definir el destino del resultado de la operación de la ALU o realizar una multiplicación con el bloque multiplicador auxiliar de apoyo la ALU.

Siguiendo el esquema del gráfico 2.4, en la fase Q1 se comienza decodificando la instrucción. En la fase Q2 se lee el registro de la memoria de datos involucrada en la instrucción. En la fase Q3 se procesa la operación en la ALU y en la fase Q4 el resultado de la operación es colocado en un registro en memoria o en el registro de trabajo W.

Gráfico 2.4. Proceso de lectura y ejecución de instrucciones



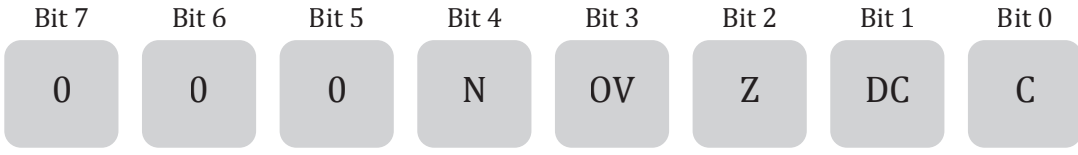
Elaboración propia.

- **Unidad aritmética lógica (ALU)**. Es la unidad central de ejecución de operaciones de procesamiento, que toma como entrada dos operandos. Uno de las entradas siempre es el registro de trabajo W de 8 bits de longitud. El otro operando puede ser un registro ubicado en alguna posición de la memoria de datos o directamente un valor literal de 8 bits de longitud. La ALU del PIC18F realiza operaciones lógicas como AND, OR, XOR, o el desplazamiento de bits y operaciones aritméticas, como suma y resta.
- **Unidad de multiplicación**. Es un circuito encargado de realizar el proceso de multiplicación de dos operandos de 8 bits de longitud. El proceso de multiplicación se ejecuta en un solo ciclo de instrucción. El resultado puede ser una cifra que requiera más de 8 bits para su representación. Por ejemplo, las cifras 200 (11001000_2) y 12 (00001100_2), ambas de 8 bits de longitud, al ser multiplicadas resultan en el valor 2400 (100101100000_2), que requiere de una longitud de 12 bits para ser representada. Los registros de 8 bits PRODH y PRODL se utilizan para almacenar la respuesta

formando un solo valor de 16 bits, que puede albergar el mayor resultado posible (11111111_2 x 11111111_2 = 1111111000000001_2).

- **Registro de estado (STATUS)**. Es un registro que está directamente asociado con el resultado de la operación de la ALU. Posee 8 bits de longitud, pero solo 5 bits tienen una representación útil. En el gráfico 2.5 se muestra una imagen del registro STATUS y de sus bits.

Gráfico 2.5. El registro de estado (STATUS)



Elaboración propia.

1. Carry (C). Es el bit LSB del registro STATUS y permite conocer si el resultado de una suma ha generado un desbordamiento. Por ejemplo, si se suman dos operandos de 8 bits, como 65 + 192, el resultado será 257. Esta cifra tiene una representación binaria de 9 bits de longitud (100000001_2). El bit MSB es '1' y no puede ser almacenado en el registro W ni en un registro de la memoria de datos, ya que ambos tienen tan solo 8 bits de longitud. De esta manera, el bit Carry se pondrá automáticamente en '1', indicando que el resultado de la operación de suma rebasó los 8 bits de longitud. El funcionamiento del bit Carry es totalmente opuesto cuando la operación realizada por la ALU es de sustracción. Si la resta de dos operandos da como resultado un número mayor o igual a cero, el bit Carry se mantendrá en estado '1'; pero si el resultado genera un desbordamiento, como en la resta de 15 – 35, el bit Carry será '0'. Este bit es de lectura y escritura, lo que significa que, si bien es cierto, cambia su estado automáticamente con el resultado de una operación, es posible cambiar su contenido de manera directa.
2. Digit Carry (DC). Este bit se encuentra en la posición 1 del registro STATUS. Opera de manera muy similar al bit Carry, con la diferencia de que representa el desbordamiento de los cuatro bits menos significativos del resultado de la operación. Este bit es muy utilizado cuando se trabajan operaciones en formato BCD.
3. Zero (Z). El bit Z se pone a '1' cuando el resultado de una operación, ya sea aritmética o lógica, es 00000000_2.
4. *Overflow* (OV). Este bit es utilizado cuando se trabaja con operandos con signo o en formato complemento a 2. Si el bit MSB del resultado cambia de estado, esto es considerado un desbordamiento u *Overflow*, ya que la respuesta pasó de un valor positivo a negativo o viceversa.
5. Negative (N). Es el bit más significativo del registro STATUS. Si el bit MSB del resultado de una operación es '1', se entiende que el resultado de dicha operación generó un valor negativo (complemento a 2). Si N es '0', se entenderá que el resultado es un valor positivo.

2.2 Las unidades de memoria

Una memoria es un circuito integrado de alta escala de integración compuesto por un conjunto de registros. Cada registro se caracteriza por tener un ancho en bits (normalmente múltiplo de 8 bits) y por contener de manera momentánea o permanente un código binario en su interior. Cada registro puede ser accedido mediante el control de un dispositivo externo para su lectura o escritura.

En el gráfico 2.6 se muestra un esquema general de una memoria. Como se puede observar, toda unidad de memoria se caracteriza por poseer un conjunto de líneas que especifican la ubicación del registro a acceder, también conocido como el bus de direcciones. Si dicho bus presenta n líneas de dirección, entonces el número total de registros o direcciones que presenta la memoria es de 2^n. Por otro lado, se tiene un bus de datos compuesto por m líneas. Por lo general, m suele ser 8, 16 o 32 bits, dependiendo del tipo de memoria, y representa la longitud del registro.

Toda memoria posee una capacidad máxima. Esta depende del bus de direcciones y del tamaño del bus de datos. Por ejemplo, una memoria de 10 líneas de dirección y 8 bits de datos tiene una capacidad total de 2^{10}x8 bits = 1024 bytes = 1 kB. Si, por ejemplo, la memoria tuviera 17 líneas de dirección y 16 bits de tamaño de registro o bus de datos, su capacidad total sería 2^{17}x16 bits = 131,072x16 bits = 128kx16 o 256 kB en total.

Adicionalmente, una unidad de memoria presenta líneas de control para definir el tipo de acceso. La línea R/W[25] permite fijar el modo de acceso de lectura o escritura. Durante la lectura, el bus de datos se convierte en un puerto de salida y permite obtener el contenido del registro apuntado por la dirección fijada en el bus de direcciones; mientras que durante la escritura, el bus de datos se convierte en un puerto de entrada y automáticamente carga el valor binario colocado en este bus en el registro apuntado por el bus de direcciones.

La línea CE[26] permite habilitar la unidad de memoria para que pueda ser accedida para un ciclo de lectura o escritura. Esta línea tiene mucha utilidad cuando se colocan múltiples memorias en paralelo y estas comparten el mismo bus de direcciones y de datos. Cuando la línea CE se pone a '0', los buses de direcciones y datos entran en estado de alta impedancia, permitiendo que la información de otra memoria pueda viajar por los buses al controlador sin afectar la información del otro dispositivo. Finalmente, con la línea OE[27] se puede habilitar el bus de datos para el proceso de lectura de los registros.

En el mundo de los sistemas embebidos existen dos tipos generales de memoria: la memoria volátil y la memoria no volátil. El primer tipo de memoria pierde la información almacenada en sus registros cada vez que el voltaje de alimentación es desconectado. El segundo tipo de memoria mantiene la información en sus registros de forma permanente, a pesar de que la tensión de alimentación haya sido desconectada.

Dentro de las memorias volátiles se encuentra la memoria de acceso aleatorio o RAM (*Random Access Memory*). Un tipo muy común que utilizan todos los microcontroladores es la memoria estática asíncrona de acceso aleatorio (SRAM). En este tipo de memorias no se requiere el uso de un reloj de sincronismo para la lectura o escritura, pero sí se necesita un orden adecuado en cada línea de control y en el bus de comunicación para que los accesos se lleven a cabo de manera satisfactoria. Las memorias

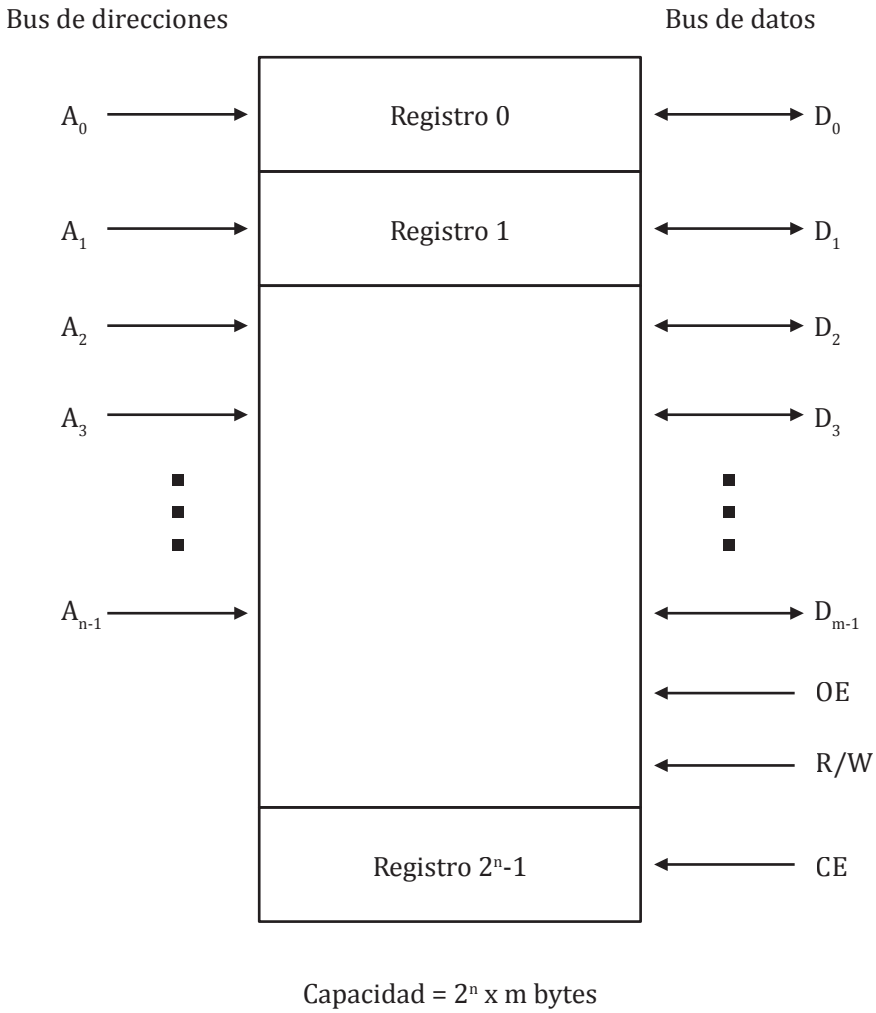
25 R/W proviene de Read/Write, que significa «lectura o escritura».

26 CE proviene de *Chip Enable*, que significa «habilitación del circuito integrado».

27 OE proviene de *Output Enable*, que significa «habilitador de salida».

SRAM permiten el acceso a cualquiera de los registros de memoria y tienen una velocidad de respuesta tanto para la lectura como para la escritura en el orden de unos cuantos nanosegundos. Usualmente, este tipo de memoria es utilizada para almacenar los datos en un sistema embebido.

Gráfico 2.6. La unidad de memoria de almacenamiento de datos



Elaboración propia.

En cuanto a las memorias no volátiles, se tienen diversos tipos. Entre los más utilizados por los sistemas embebidos actuales existen dos: la memoria EEPROM[28] y la memoria FLASH. El primer tipo de memoria se caracteriza por tener un comportamiento muy similar a las memorias SRAM (aunque algunas memorias EEPROM utilizan buses de datos seriales en vez de paralelos) y tener capacidades limitadas (en el orden de unos cuantos kB). El proceso de borrado y escritura es completamente eléctrico, con lo cual no se requiere de algún tipo de elemento adicional para completar estos eventos (como en el caso de las memorias EPROM[29]). Las memorias EEPROM poseen ciclos de lectura bastante rápidos, pero sus ciclos de escritura suelen ser lentos, demorando un tiempo en el orden de unos cuantos milisegundos para la escritura de cada registro. Por otra parte, las memorias FLASH son diseños mejorados

28 *Electrically Erasable Read Only Memory*.

29 Esta memoria requiere de luz ultravioleta para llevar a cabo el proceso de borrado.

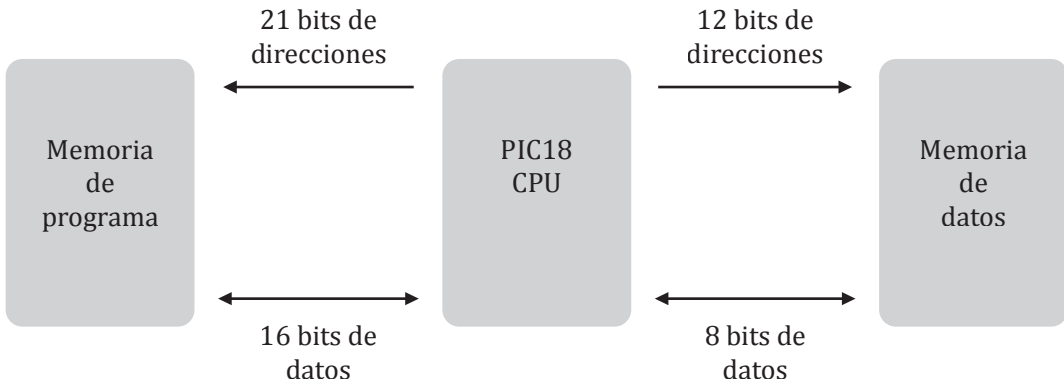
de las memorias EEPROM. Se caracterizan por poseer capacidades más elevadas y ser borradas y escritas de forma eléctrica. El proceso de escritura requiere el acceso por bloques o sectores de memoria, de manera que un grupo de registros se graben en un solo acceso a escritura, lo cual hace que este proceso sea mucho más veloz que en el caso de las memorias EEPROM. Adicionalmente, la memoria FLASH posee una alta durabilidad para los ciclos de borrado y grabado (más de 10 000 ciclos) y actualmente es el tipo de memoria más utilizada en múltiples dispositivos comerciales. En un sistema embebido, la memoria FLASH es utilizada como memoria de programa y de datos permanentes.

Los microcontroladores PIC18F poseen cuatro tipos de unidades de memoria:

- La memoria de datos
- La memoria de programa
- La pila
- La memoria EEPROM

El PIC18 posee una arquitectura Harvard, es decir, la memoria de datos y de programa se conectan al CPU con buses de direcciones y datos independientes, lo cual facilita el acceso a las instrucciones y datos en paralelo. En el gráfico 2.7 se observa el esquema de conexión entre el procesador y las unidades de memoria en el PIC18.

Gráfico 2.7. La organización de memoria en el PIC18



Elaboración propia.

2.2.1 La memoria de datos

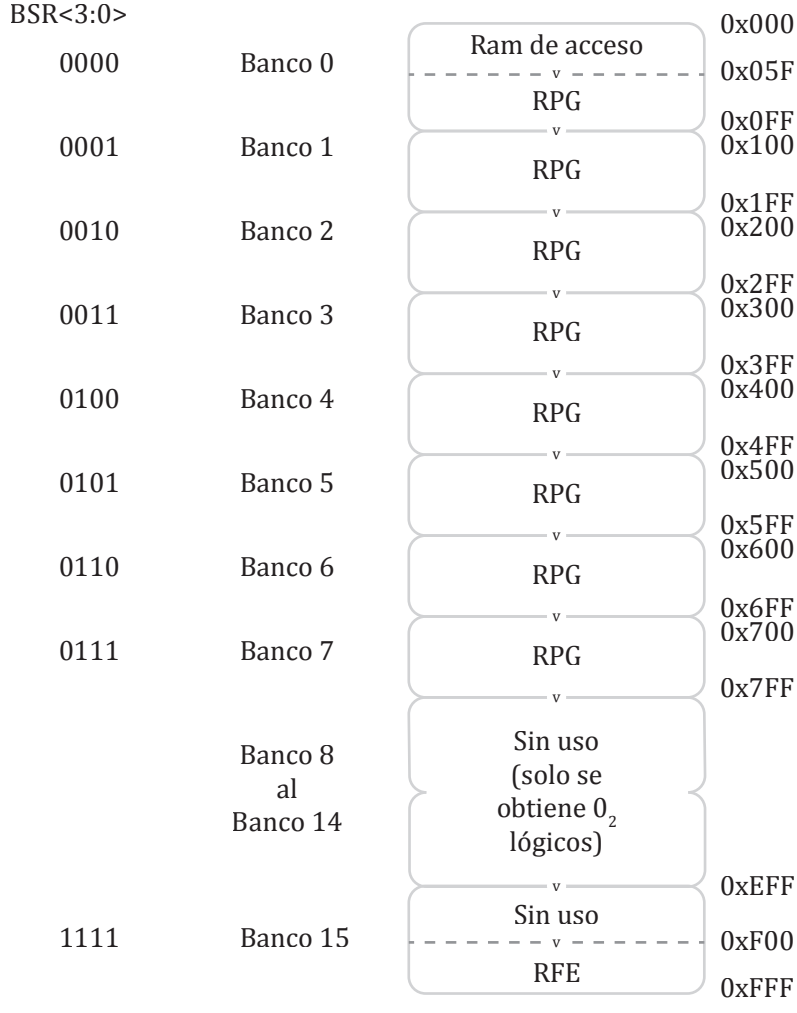
La memoria de datos del PIC18F es del tipo RAM estática (SRAM). Cada registro en la memoria de datos tiene una dirección de 12 bits, permitiendo el direccionamiento de 4096 (2^{12}) registros de 1 byte de longitud. El espacio de memoria está dividido en 16 bancos que contienen 256 bytes cada uno. Los PIC18F2455/2550/4455/4550 implementan 8 bancos completos, para un total de 2048 bytes. En el gráfico 2.8 se muestra la organización de la memoria de datos para estos dispositivos.

La memoria de datos contiene registros de función especial (RFE) y registros de propósito general (RPG). Los RFE son usados para el control y verificación del estado de las funciones de los periféricos, mientras que los RPG son usados para almacenamiento de datos y operaciones específicas en la aplicación del usuario. Si se hace un lectura en aquellas direcciones no implementadas (bancos 8 al 14) siempre se obtendrá una lectura de 0x00 en cada registro.

El PIC18F divide su memoria de datos en bancos para poder ofrecer un mayor número de modos de acceso y direccionamiento de memoria. Es posible manipular la SRAM de acceso simplemente definiendo la dirección absoluta de un registro indicando su valor de 12 bits. Sin embargo, esto no es preferible cuando se desea un acceso rápido, ya que para manipular un valor de 12 bits, el procesador requiere de dos ciclos de instrucción.

Para facilitar el acceso a la memoria en un solo ciclo de instrucción se puede acceder a cada uno de los bancos que contienen una dirección relativa de 8 bits. Por ejemplo, el banco 1 se encuentra ubicado entre las direcciones 0x100 y 0x1FF. Entre estas direcciones se tienen 256 registros o bytes que pueden ser direccionados con un solo registro de dirección de 8 bits. Para esto, existe un registro selector de banco denominado BSR[30], el cual contiene en sus cuatro bits menos significativos la combinación binaria que permite especificar el banco a acceder. Luego, para seleccionar la dirección de 8 bits relativa al banco, se usa un operando de la instrucción de acceso a memoria.

Gráfico 2.8. Mapa de memoria de datos del PIC18F2455/2550/4455/4550



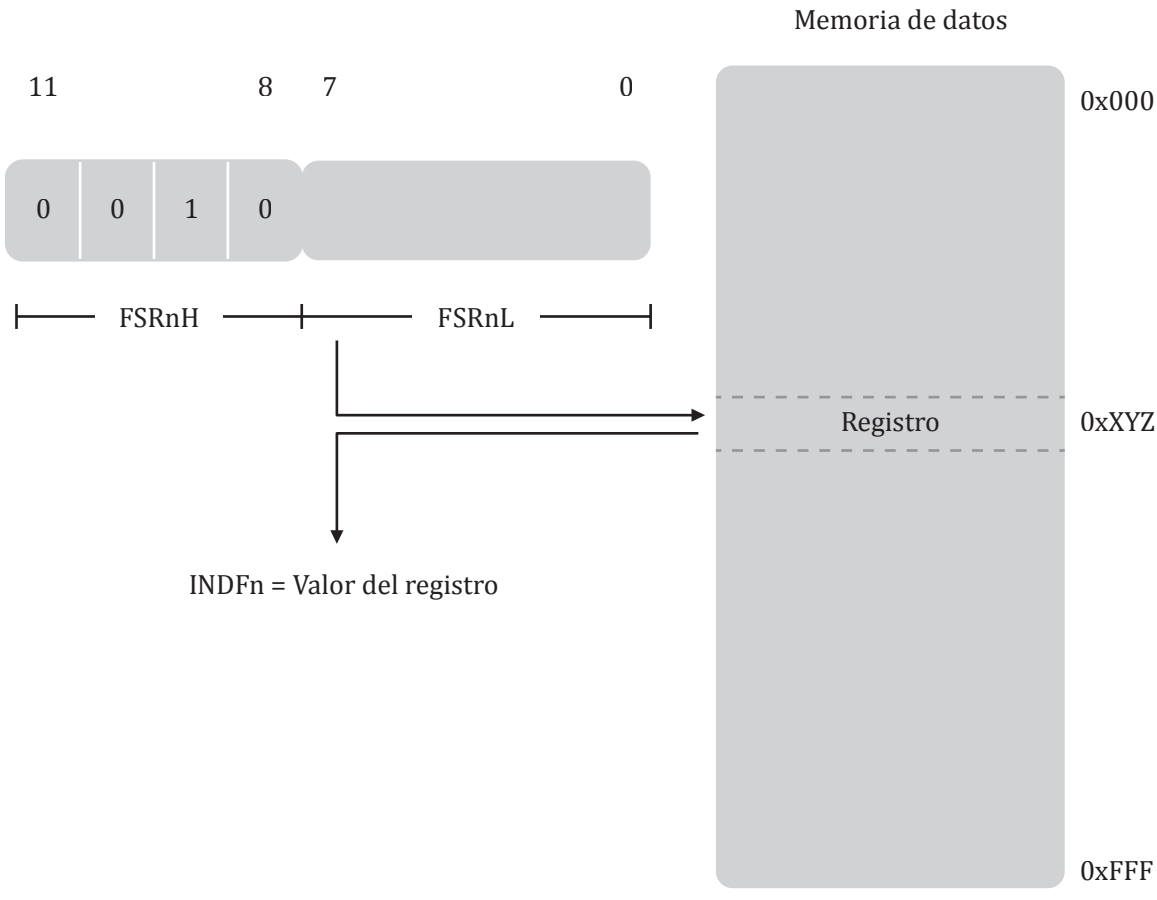
Elaboración propia.

30 Del inglés *Bank Select Register*.

Para facilitar el acceso a la memoria en un solo ciclo de instrucción se puede acceder a cada uno de los bancos que contienen una dirección relativa de 8 bits. Por ejemplo, el banco 1 se encuentra ubicado entre las direcciones 0x100 y 0x1FF. Entre estas direcciones se tienen 256 registros o bytes que pueden ser direccionados con un solo registro de dirección de 8 bits. Para esto, existe un registro selector de banco denominado BSR, el cual contiene en sus cuatro bits menos significativos la combinación binaria que permite especificar el banco a acceder. Luego, para seleccionar la dirección de 8 bits relativa al banco, se usa un operando de la instrucción de acceso a memoria.

Por otro lado, se cuenta con tres registros de función especial de direccionamiento indirecto como son FSR0, FSR1 y FSR2 de 12 bits cada uno. Cada uno de estos registros se divide en dos partes: FSRxH y FSRxL, donde x puede ser 0, 1 o 2 dependiendo del registro que se quiere usar. La parte alta del registro FSRxH contiene los cuatro bits más significativos de la dirección, mientras que la parte baja FSRxL contiene los 8 bits menos significativos. El valor de cada registro FSRxL apunta a una dirección de 8 bits dentro del banco seleccionado. El contenido de cada dirección puede ser modificado haciendo uso de los registros INDF0, INDF1 e INDF2, los cuales pueden ser leídos o escritos. En el gráfico 2.9 se observa un esquema del modo de direccionamiento indirecto de los bancos de memoria haciendo uso de estos registros.

Gráfico 2.9. Modo de direccionamiento indirecto haciendo uso de los bancos de memoria



Elaboración propia.

El BSR cumple con la misma función que el registro FSRxH, con la diferencia de que el modo de acceso a una dirección relativa es mediante un valor de 8 bits, permitiendo el acceso a una dirección de 12 bits de memoria de manera indirecta mediante un solo ciclo de instrucción. Solo los cuatro bits menos significativos del BSR son implementados (BSR3:BSR0). Los cuatro bits más significativos no tienen uso; siempre serán '0' y no pueden ser escritos (son bits de solo lectura). El BSR puede ser cargado directamente usando la instrucción *movlb*.

Los bancos 4 al 7 de la memoria de datos están interconectados a un puerto RAM dual especial. Cuando el módulo USB en los PIC18F2550/4550 está deshabilitado, los registros en estos bancos son usados como un simple espacio de memoria de datos. Cuando el módulo USB está activo, la memoria en estos bancos es tratada como un buffer para la operación USB. Esta área se comparte entre el procesador del PIC18 y el motor de interfaz serial USB y es usado para transferir datos directamente entre los dos.

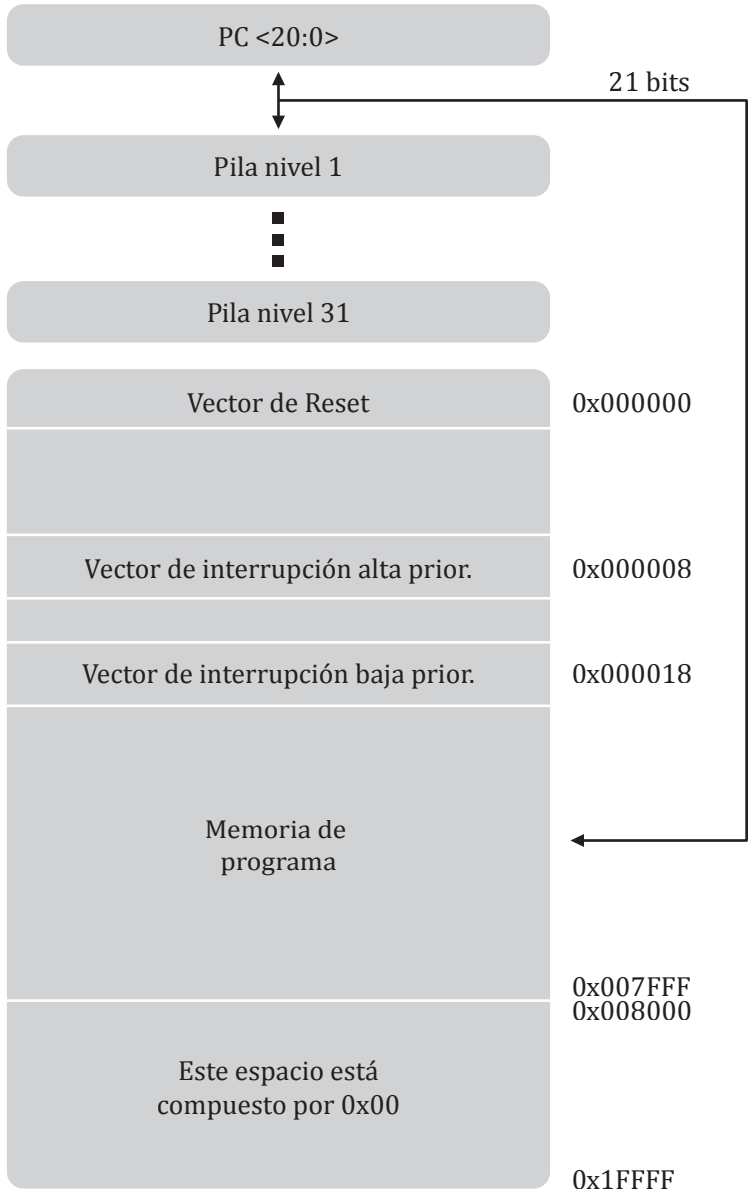
2.2.2 La memoria de programa

La memoria de programa o memoria FLASH puede ser leída, escrita y borrada durante la operación normal del microcontrolador. Una lectura de la memoria de programa es ejecutada sobre un registro a la vez. Una escritura de la memoria de programa es ejecutada en bloques de 32 registros a la vez. La memoria de programa se borra en bloques de 64 registros en simultáneo.

Cada registro de la memoria FLASH contiene 16 bits de longitud. En total, la memoria cuenta con un bus de direcciones de 21 bits de los cuales el bit LSB siempre es '0', por lo cual es posible direccionar hasta 2^{20}(1M x 2 bytes) direcciones o registros. Los registros de la memoria FLASH tienen diversos propósitos: pueden ser usados para almacenar instrucciones del programa, datos constantes y también se tienen posiciones de la memoria específicas para el manejo de interrupciones de alta y baja prioridad, configuración del microcontrolador y vector de Reset.

No todas las direcciones de la memoria FLASH están habilitadas para ser utilizadas por el usuario. Por ejemplo, el PIC18F4550 posee solo 32 kB de capacidad. Esto implica que muchas de las direcciones de acceso no se encuentran habilitadas, tal y como se muestra en el esquema de la memoria FLASH del gráfico 2.10.

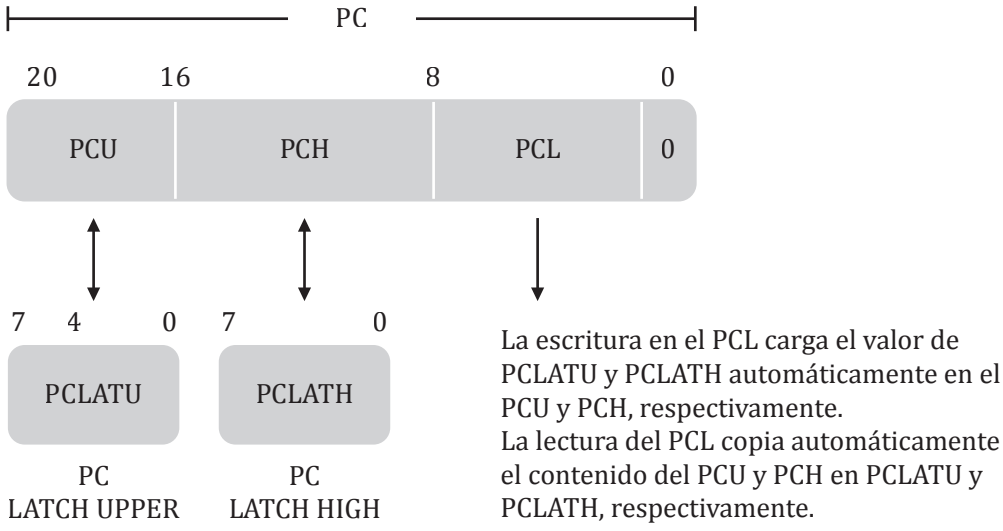
Gráfico 2.10. Organización de la memoria de programa en el PIC18F4550



Elaboración propia.

El contador de programa (PC) especifica la dirección de la instrucción cargada por la unidad de control para su ejecución. El PC posee 21 bits de ancho y se divide en tres registros separados de 8 bits. El byte bajo, conocido como el registro PCL, es tanto de lectura como de escritura, y su bit LSB siempre es '0', con lo cual el PC siempre apunta a direcciones pares. El byte alto o el registro PCH contiene los bits PC<15:8>, y no es posible leer o escribir sobre este directamente. Para actualizar el registro PCH se utiliza el registro PCLATH. El byte más alto del PC se denomina PCU. Este registro contiene los bits PC<20:16>, y tampoco es de escritura y lectura directa. Para actualizar el PCU se requiere un acceso al registro PLCATU. En el gráfico 2.11 se muestra un esquema de los registros relacionados a la operación con el PC.

Gráfico 2.11. Registros relacionados con la operación del PC



Elaboración propia.

El vector de Reset es la primera dirección de la memoria FLASH (0x00000) y es la posición por defecto que el PC apunta cada vez que el microcontrolador acaba de ser energizado o se ha realizado un evento de Reset. En esta dirección debe estar guardada una instrucción de salto que se encargue de llevar al PC a una dirección por encima del valor 0x00018 (vector de interrupción de baja prioridad) para que, a partir de este punto, se ejecute el código del usuario.

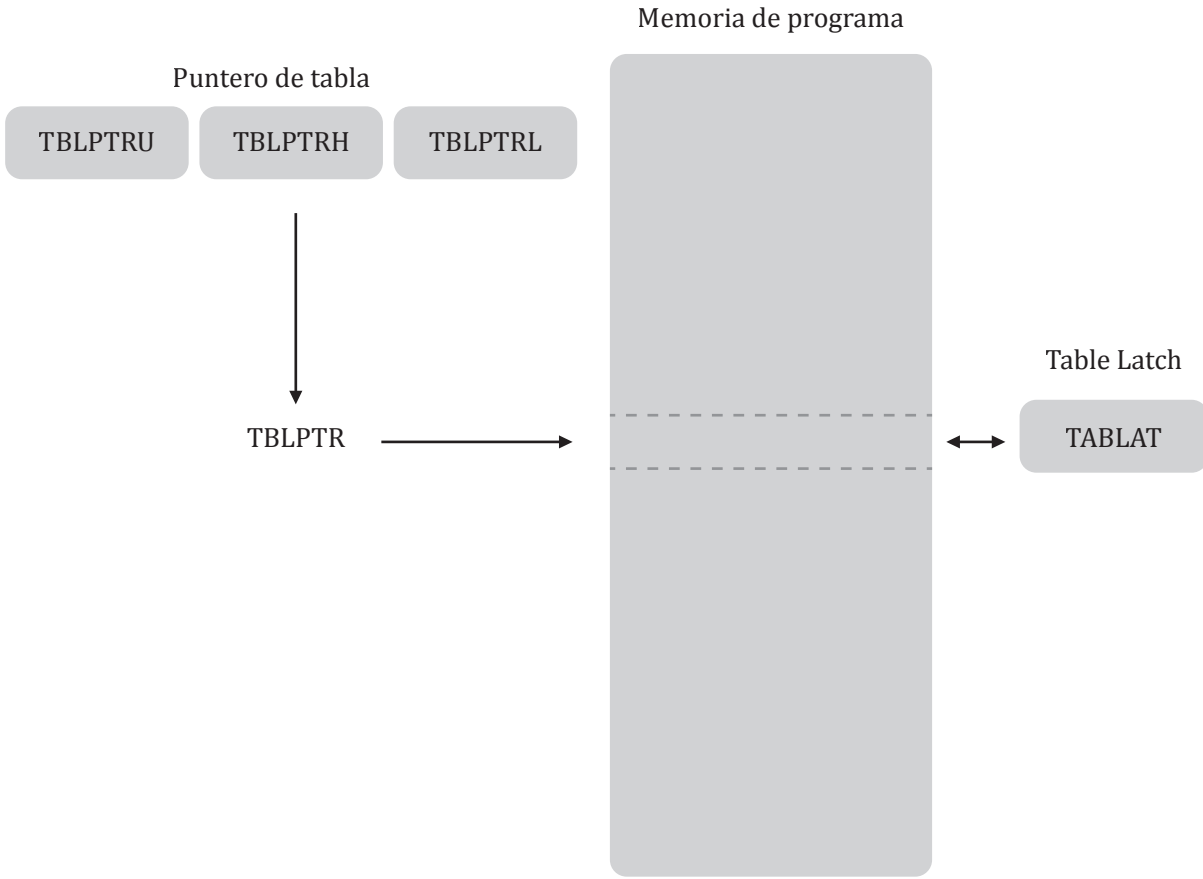
Otra ubicación especial es el vector de interrupción de alta prioridad que se encuentra en la dirección 0x00008, y es la ubicación que toma el PC cada vez que ocurre una interrupción de alta prioridad. De la misma manera, la dirección 0x00018 es la ubicación que toma el PC cuando ocurre una interrupción de baja prioridad. En ambos casos, se debe almacenar una instrucción *goto dirección* que permita llevar el PC a una dirección donde se tenga almacenada la rutina de servicio de interrupción. Este tema se tocará con más detalle en el capítulo 6.

Por lo general, la zona de ejecución de instrucciones debe estar por debajo de los vectores de interrupción. Es posible utilizar la dirección 0x0001A para este fin. Se debe tener cuidado, porque no todas las posiciones de la memoria FLASH pueden ser utilizables. Existe una región entre las direcciones 0x0001A y 0x7FFF que puede ser usada para el almacenamiento de instrucciones y datos. Mientras la región entre las direcciones 0x8000 y 0x1FFFFF se encuentra inutilizada. De esta manera, al ejecutar la instrucción de salto se deberá limitar la dirección de acceso a los rangos permitidos por el mapa de memoria de programa.

La memoria FLASH también puede ser utilizada para almacenar datos constantes, como tablas de mensajes con caracteres ASCII. Esto es muy usual en aplicaciones que requieren mostrar mensajes en un módulo LCD alfanumérico o en un puerto serie para transmisión de datos a un computador o a un módem. Estos datos se guardan en una región útil de la memoria de programa al mismo tiempo que se instalan las instrucciones de acceso a las tablas en el proceso de grabación. Las tablas de datos son simples arreglos binarios colocados en registros de la memoria FLASH con direcciones ordenadas de forma ascendente. Es posible también grabar información en la memoria de programa durante tiempo de ejecución, haciendo uso de registros especiales del RFE. Para poder tener acceso a la lectura de información de las tablas se pueden utilizar unos registros especiales, como el TBLRD (Table Read), y para la escritura, el registro TBLWT (Table Write).

Si se analiza el esquema del gráfico 2.12, existe un registro conocido como el puntero de tabla (registro TBLPTR[31]), que se encuentra compuesto por tres registros separados: TBLPTRU, TBLPTRH y TBLPTRL, cada uno de estos de 8 bits. El TBLPTR apunta a la dirección de 21 bits de la tabla que se desea leer o escribir en la memoria de programa. Los tres registros que conforman el TBLPTR se encuentran implementados en la región RFE de la memoria de datos. El registro TABLAT apunta al contenido en cada dirección fijada en el puntero de tabla, de una manera similar al caso de los registros INDFx y FSRx en la memoria de programa.

Gráfico 2.12. Esquema del proceso de direccionamiento de tabla para lectura y escritura en la memoria de programa



Elaboración propia.

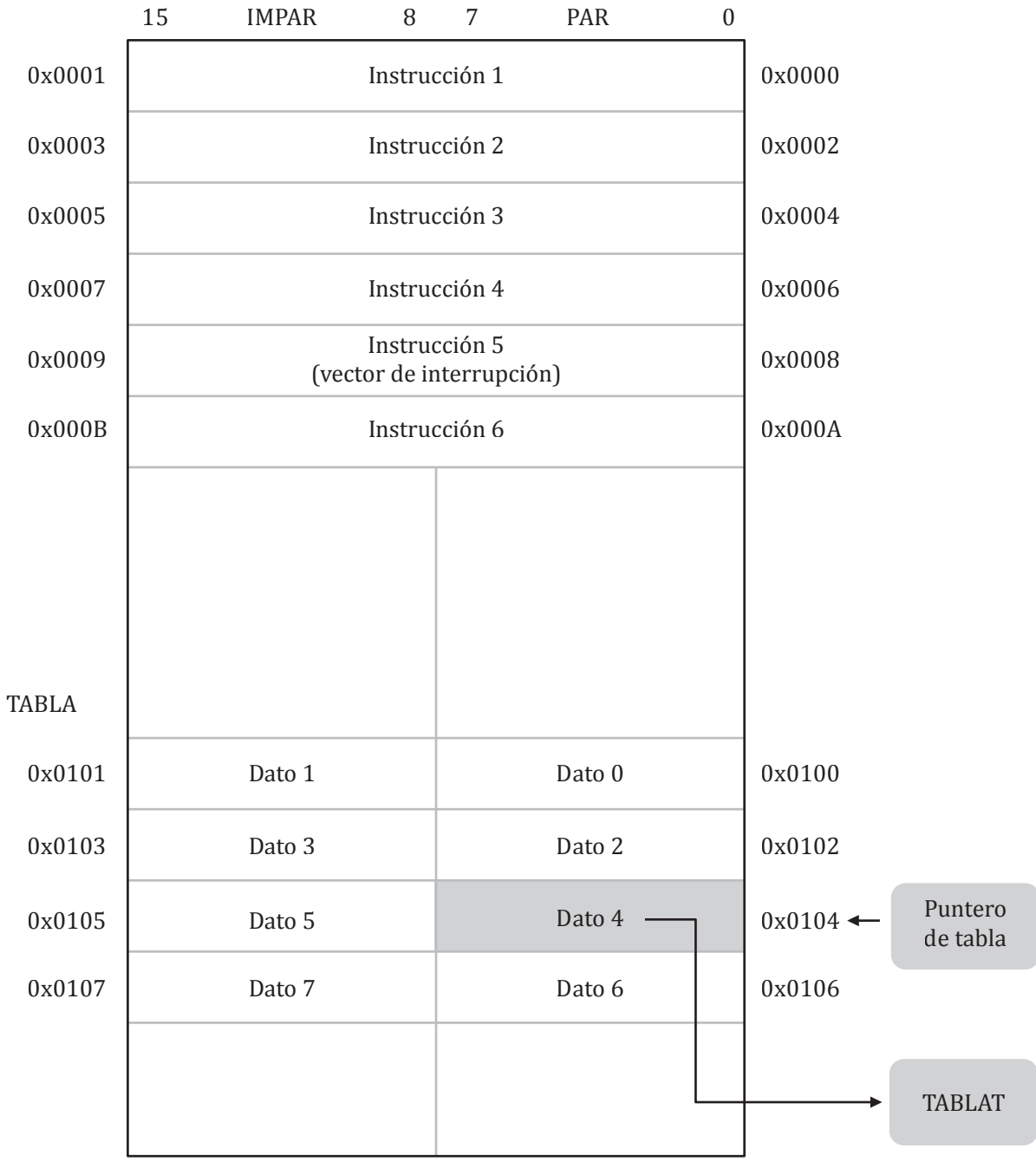
La ventaja del uso del puntero de tabla es que este registro es de 21 bits y sí puede hacer uso del bit LSB, a diferencia del PC. De esta manera, la memoria de programa se divide en dos regiones: una de dirección impar y otra de dirección par. El registro TABLAT solo puede apuntar a 8 bits, por tanto, la dirección par de TBLPTR apunta al byte bajo de un registro, mientras que la dirección impar de TBLPTR apunta al byte alto del mismo registro. Este esquema se observa en el gráfico 2.13. Con esto se puede aprovechar todo el espacio de la memoria de programa para el almacenamiento de bytes. En otras palabras, se accede a la memoria de programa como si fuera de 8 bits de longitud.

31 Del inglés *Table Pointer.*

2.2.3 La pila

La pila (Stack) es una memoria SRAM de 31 registros y 21 bits de ancho, y se encuentra posicionada entre la memoria de programa y el PC. Para tener acceso a los registros de la pila se utiliza un puntero de pila de 5 bits de nombre STKPTR[32]. El puntero de pila se puede leer y escribir, y la dirección a la que apunta el puntero se puede manipular a través de los registros Top-of-Stack. Es posible introducir datos o leerlos en la pila haciendo uso de estos registros.

Gráfico 2.13. Proceso de almacenamiento de datos de una tabla en la memoria de programa



Elaboración propia.

32 Del inglés *Stack Pointer*.

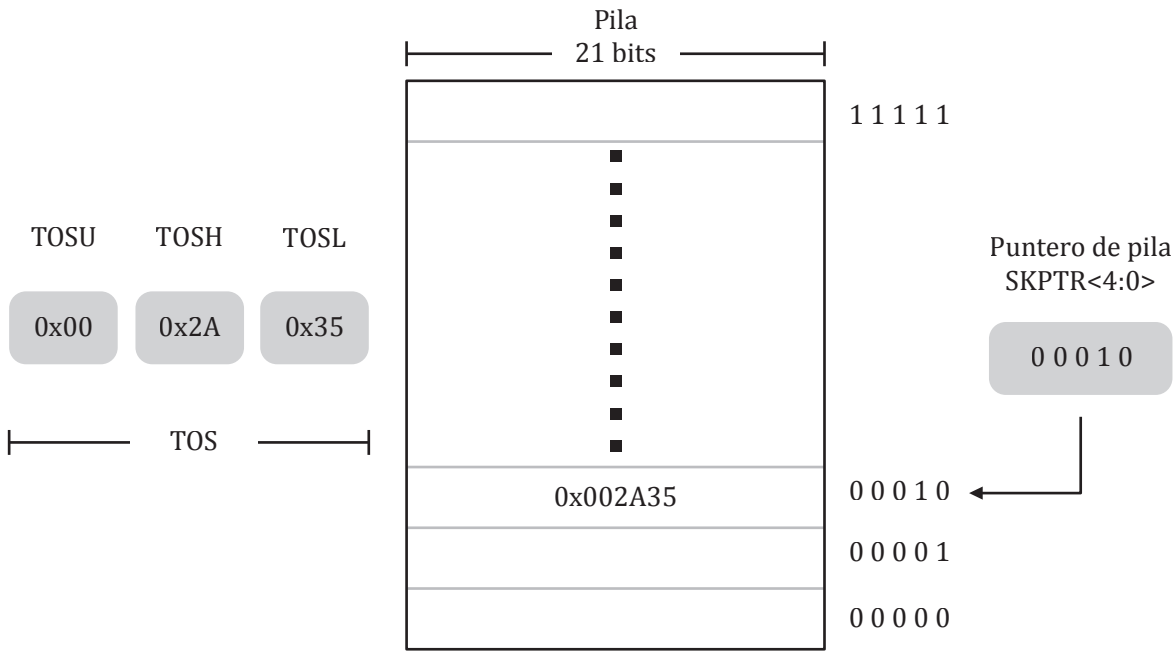
La función principal del puntero de pila es la de almacenar los valores del PC cada vez que se ejecuta una instrucción del tipo *call* (instrucción que causa que el valor del PC cambie su contenido por más de dos dígitos, lo cual se conoce como instrucción de salto), que origina automáticamente el almacenamiento de la dirección del PC+2 en la pila antes de que tome su nuevo valor de salto (lo que se conoce como una operación *push*). El puntero de pila primero se incrementa a la ubicación apuntada y luego el contenido de esta dirección se escribe con el valor del PC (que apunta a la dirección anterior +2 después del *call*). Una instrucción tipo *Return* causa que el contenido de la ubicación apuntada por el puntero de pila sea transferido al PC y luego el puntero de pila sea restado en 1 (lo que se conoce como una operación POP).

El puntero de pila se inicializa en la dirección 0x0000 después de un evento de Reset. No existen registros RFE asociados a la ubicación correspondiente del puntero de pila, pero sí existen bits de estado que indican si la pila se ha sobrepasado por operaciones PUSH o POP.

La parte alta de la dirección de la pila (TOS[33]) es de lectura o escritura. Un conjunto de tres registros TOSU:TOSH:TOSL mantiene el contenido de la ubicación de la pila apuntado por el registro STKPTR. Esto permite al programador acceder al contenido de la pila de forma directa. Después de las instrucciones *call*, *rcall* o una interrupción, el *firmware* puede leer o escribir un valor en el TOS accediendo a los registros TOSU:TOSH:TOSL. Al momento de retornar un valor al PC, el microprocesador retorna el valor desde los registros TOSU:TOSH:TOSL. Esto se muestra en el esquema del gráfico 2.14.

El registro RFE de nombre STKPTR contiene la posición del valor apuntado en la pila y a los bits STKFUL[34] y el bit STKUNF[35], que corresponden a las posiciones 7 y 6 del registro, respectivamente. La dirección del puntero de pila puede tomar los valores 0 a 31. El puntero de pila se incrementa antes de que los valores sean puestos en la pila y se autodecrementa después de que los valores son extraídos de la pila. En un evento de Reset, el puntero de pila toma el valor cero.

Gráfico 2.14. Registros asociados al manejo de la pila



Elaboración propia.

33 Del inglés *Top of the Stack,* que corresponde al registro que alberga el contenido de 21 bits de la pila.

34 Del inglés *Stack Full*, que significa «pila llena».

35 Del inglés *Stack Underflow*, que significa «pila sobrevaciada».

Después de que el PC ha sido empujado o almacenado dentro de la pila 31 veces (sin ejecutar la instrucción POP ninguna vez), el bit STKFUL se pone a '1', indicando que la pila se encuentra llena y que una instrucción PUSH adicional hará perder el valor contenido en la dirección 0000_2.

La respuesta que se lleva a cabo después de que la pila se llena depende del estado del bit STVREN[36]. Este bit se encuentra en el registro CONFIG4L, que pertenece a una dirección especial de la memoria de programa (0x300006) y contiene los bits de configuración de los modos de funcionamiento del microcontrolador. Si el bit STVREN es '1' (estado por defecto), la operación de *push* número 31 meterá el PC en la pila, pondrá el bit STKFUL en '1' y reiniciará al microcontrolador (Reset). El bit STKFUL permanecerá en '1' y el puntero de pila será '0'.

Si el bit STVREN es '0', el bit STKFUL se pondrá en '1' al ocurrir la operación de PUSH número 31 y el puntero de pila se incrementará a 31. Cualquier *push* adicional no sobrescribirá al *push* número 31 y el STKPTR permanecerá en 31, lo cual implica que se inhibirán las escrituras en la pila.

Cuando una operación de *pop* se ha realizado muchas veces para descargar la pila, el siguiente *pop* retornará un valor '0' al PC y pondrá el bit STKUNF en '1', mientras que el puntero de pila se mantendrá en '0'. El bit STKUNF permanecerá en '1' hasta que sea borrado por *firmware*. El registro STKPTR se muestra en el gráfico 2.15.

Gráfico 2.15. El registro de estado y direccionamiento del puntero de pila

STKPTR (Registro puntero de pila)

7	6	5	4	3	2	1	0
STKFUL	STKUNF	———	SP4	SP3	SP2	SP1	SP0

STKFUL
1: Ocurrió un desbordamiento de la pila por sobrellenado
0: No ocurrió ningún desbordamiento
STKUNF
1: Ocurrió un subdesbordamiento de la pila por sobrevaciado
0: No ocurrió el subdesbordamiento
SP4-SP0: Ubicación del puntero de pila

Elaboración propia.

2.2.4 La memoria EEPROM

La memoria EEPROM es no volátil y se encuentra separada de la memoria de datos y la de programa; por lo tanto, tiene sus buses de comunicación independientes. Se utiliza para el almacenamiento a largo plazo de datos del programa, y cualquier dirección de esta puede ser accedida para el proceso de lectura y escritura durante el tiempo de ejecución del programa.

36 Del inglés *Stack Overflow Reset Enable,* que corresponde a un evento de Reset generado por el desboramiento de la pila.

La memoria EEPROM no se encuentra dentro del mapa de memoria de datos del microcontrolador, pero es direccionada indirectamente a través de ciertos RFE. La capacidad de la EEPROM en cualquier PIC18 es de unos cuantos bytes. En el caso específico del PIC18F4550, contiene 256 bytes.

Se utilizan cuatro RFE para leer y escribir la EEPROM, al igual que en la memoria de programa. Estos registros son los siguientes:

- EECON1
- EECON2
- EEDATA
- EEADR

La EEPROM permite la lectura y escritura por byte. Cuando se interconecta a un bloque de memoria, el registro EEDATA mantiene los 8 bits de datos para el proceso de lectura/escritura, y EEADR mantiene la dirección de la ubicación de la EEPROM que se está accediendo.

La memoria EEPROM está diseñada para tener una duración muy alta, con muchos ciclos de borrado y grabado. Una escritura de un byte automáticamente borra la ubicación y sobreescribe el nuevo dato (se borra el registro antes de la escritura). El tiempo de escritura es controlado por un temporizador dentro del circuito integrado.

El acceso a la EEPROM se logra a través de dos registros: EECON1 y EECON2. Estos son los mismos registros con los cuales se controla el acceso a la memoria de programa y son usados de la misma manera para la EEPROM. Más adelante, en los siguientes capítulos, se mostrarán ejemplos para la lectura y escritura de datos en la memoria EEPROM del PIC18F4550.

2.3 Almacenamiento de las instrucciones en la memoria de programa

Normalmente, las instrucciones que la Unidad de control ejecuta de manera secuencial desde cada dirección de la memoria de programa han tenido que ser escritas en un documento de texto por un programador. El código binario con las instrucciones que se encontrarán almacenadas en cada dirección de la memoria de programa se denomina «código máquina». Cada instrucción está compuesta por una combinación de 16 bits que va a ser interpretada por el decodificador de instrucciones del procesador.

Para el programador el código máquina resulta muy complejo, ya que programar usando secuencias binarias no solo puede cansar la vista, sino también ser una fuente grande de errores involuntarios en la programación. Para evitar esto, hace muchos años los fabricantes de microcontroladores diseñaron compiladores para el lenguaje ensamblador que permiten que el programador escriba mnemónicos, que son códigos de texto que reemplazan al código de máquina. Estos mnemónicos son un conjunto de instrucciones en lenguaje ensamblador que el compilador puede transformar a su equivalente en código máquina de 16 bits de longitud. Para el programador resulta mucho más práctico y menos tedioso aprender a programar haciendo uso de mnemónicos. Un mnemónico típico para el lenguaje ensamblador del PIC18 puede ser *movlw 0x30*. Esto significa cargar en el registro W del procesador con el valor literal 0x30. Esto tiene una traducción en lenguaje de máquina, que es la siguiente: 0000111000000011_2. Obviamente, se entiende que el uso del mnemónico simplifica enormemente la comprensión del programa.

Según la arquitectura del PIC18, se cuenta con un total de 77 instrucciones. Cada una de ellas tiene su equivalente en mnemónico para lenguaje ensamblador. Toda instrucción se caracteriza por tener un código de operación u *Opcode*[37], que es un código binario que define el tipo de instrucción. Por ejemplo, movlw 0x80 y addlw 0x20 son instrucciones diferentes. La primera representa la operación de cargar el valor literal 0x80 en el registro de trabajo W y la segunda representa una operación de suma. De esta manera, cada una de estas instrucciones tiene un *Opcode* distinto. Adicionalmente, el valor 0x80 de la primera instrucción y 0x20 de la segunda representan argumentos que son parte también de la instrucción (en el primer caso 0x80 es el valor a guardar en el registro W y en el segundo caso, 0x20 es el valor que se va a sumar con el contenido de W).

Existen en total cinco grupos generales de instrucciones que se codifican de manera diferente en el código máquina. Estos grupos son los siguientes:

Instrucciones orientadas a operaciones de bytes

El formato del código de máquina generado se muestra en el gráfico 2.16. Aquí, el código de operación está compuesto por 6 bits. Este código especifica una operación realizada por la ALU. El bit *d* permite definir el destino. Como se podrá recordar del gráfico 2.3, la ALU puede guardar el resultado de una operación en dos ubicaciones diferentes: el registro W o una dirección de la memoria de datos. Si *d* es '1', el resultado de la operación se almacena en una dirección de la memoria SRAM. En caso de sea '0', el resultado se almacenará en el registro W.

Gráfico 2.16. Formato del código de máquina para las instrucciones orientadas a operaciones con bytes

Elaboración propia.

El bit *a* permite definir si el acceso a la memoria de datos es a través del registro BSR o usando la dirección absoluta de la SRAM. Si *a* es '1', se estará usando el banco apuntado por BSR, mientras que si *a* es '0', entonces se utilizará la dirección absoluta (usando una de las 256 direcciones del banco 0). El valor de *f* es la dirección del registro en memoria de datos y su valor será interpretado dependiendo del bit *a*.

En el gráfico 2.17 se muestran todas las instrucciones en lenguaje ensamblador para PIC18 orientadas a operaciones con bytes.

37 Del inglés *Operation Code*.

Gráfico 2.17. Instrucciones orientadas a operaciones con bytes

Mnemónico	Descripción
ADDWF f,d,a	Sumar el registro W con f
ADDWFC f,d,a	Sumar el registro W más el bit Carry con f
ANDWF f,d,a	Operación AND entre el registro W y f
CLRF f,a	Cargar el registro f con 0x00
COMF f,d,a	Obtener el complemento de f
CPFSEQ f,a	Comparar f con el registro W y saltar si son iguales
CPFSGT f,a	Comparar f con el registro W y saltar si f es mayor
CPFSLT f,a	Comparar f con el registro W y saltar si f es menor
DECF f,d,a	Restar f en 1
DECFSZ f,d,a	Restar f en 1, saltar si es 0x00
DCFSNZ f,d,a	Restar f en 1, saltar si no es 0x00
INCF f,d,a	Incrementar f en un dígito
INCFSZ f,d,a	Incrementar f en un dígito, saltar si es 0x00
INFSNZ f,d,a	Incrementar f en un dígito, saltar si no es 0x00
IORWF f,d,a	OR inclusiva entre el registro W y f
MOVF f,d,a	Mover el contenido de f al destino d
MOVWF f,a	Mover el registro W a f
MULWF f,a	Multiplicar el registro W con f
NEGF f,a	Negar f
RLCF f,d,a	Rotar f a la izquierda 1 bit y cargar el bit MSB con el Carry
RLNCF f,d,a	Rotar f a la izquierda 1 bit (sin Carry)
RRCF f,d,a	Rotar f a la derecha 1 bit y cargar el bit LSB con el Carry
RRNCF f,d,a	Rotar f a la derecha (sin Carry)
SETF f,a	Poner f a 0xFF
SUBFWB f,d,a	Restar f del registro W y el bit Borrow
SUBWF f,d,a	Restar el registro W de f
SUBWFB f,d,a	Restar el registro W de f y el bit Borrow
SWAPF f,d,a	Intercambiar nibbles
TSTFSZ f,a	Analizar f y saltar si es 0x00
XORWF f,d,a	OR Exclusiva entre el registro W y f

Elaboración propia.

Por ejemplo, si se elige la instrucción addwf 0x03,0,0, se está generando el siguiente código de máquina:

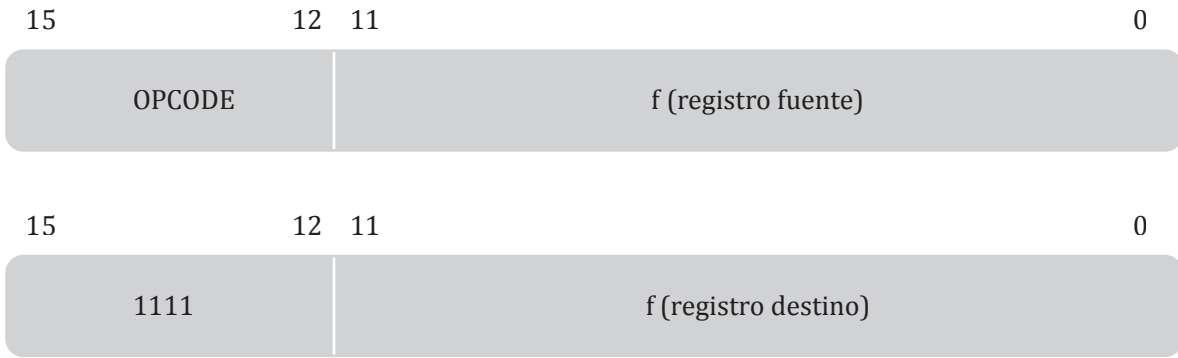
OPCODE: 001001_2
d: '0'
a: '0'
f: 00000011_2

Entonces, el código máquina generado es 0010010000000011_2.

Instrucciones orientadas a operaciones byte a byte

Para este tipo de instrucciones, el formato del código máquina se muestra en el gráfico 2.18. Aquí solo existe un único mnemónico que cumple con esta categoría, el cual es movff fs,fd y representa la instrucción que indica que el contenido del registro definido en la dirección fs (fuente) debe copiarse a la dirección fd (destino). Ambas direcciones son absolutas, por lo cual son valores de 12 bits. El código máquina generado entrega dos valores de 16 bits; así, la ejecución de esta instrucción requiere de dos ciclos de instrucción.

Gráfico 2.18. Formato del código de máquina para la instrucción movff



Elaboración propia.

Aquí el valor del *Opcode* es 1100_2. Después, los 12 bits restantes especifican la dirección absoluta de origen para el primer código binario y luego la dirección absoluta del destino para el segundo código.

Instrucciones orientadas a la manipulación de bits en registros

En el gráfico 2.19 se observa el formato para este tipo de instrucciones. En este tipo de mnemónicos, el *Opcode* presenta solo 4 bits. Aparece un nuevo campo *b* de tres bits, que permite especificar la posición del bit en el registro (posición 0 a 7). Luego, el bit *a* sigue especificando si el campo *f* se interpreta como dirección absoluta o relativa al banco seleccionado.

Gráfico 2.19. Formato del código máquina para instrucciones orientadas a manipulación de bits

Elaboración propia.

En el gráfico 2.20 se muestran todas las instrucciones orientadas a la manipulación de bits de los registros de la memoria de datos.

Gráfico 2.20. Instrucciones orientadas a operaciones con bits

Mnemónico		Descripción
BCF	f,b,a	Bit b de f en '0'
BSF	f,b,a	Bit b de f a '1'
BTFSC	f,b,a	Analizar el bit b de f y saltar si '0'
BTFSS	f,b,a	Analizar el bit b de f y saltar si es '1'
BTG	f,b,a	Negar el estado del bit b de f

Elaboración propia.

Instrucciones con operaciones literales

En el gráfico 2.21 se observa el formato para el tipo de instrucción literal. En este caso ya no se utiliza como segundo parámetro una dirección relativa o absoluta, sino más bien un valor literal, el cual va a interactuar con el destino de la operación, que, por lo general, pueden ser los registros W, FSR o BSR.

Gráfico 2.21. Formato del código máquina para instrucciones con operaciones literales

K = valor inmediato de 8 bits

Elaboración propia.

En el gráfico 2.22 se muestran todas las instrucciones con operaciones literales para el lenguaje ensamblador del PIC18.

Gráfico 2.22. Instrucciones con operaciones literales

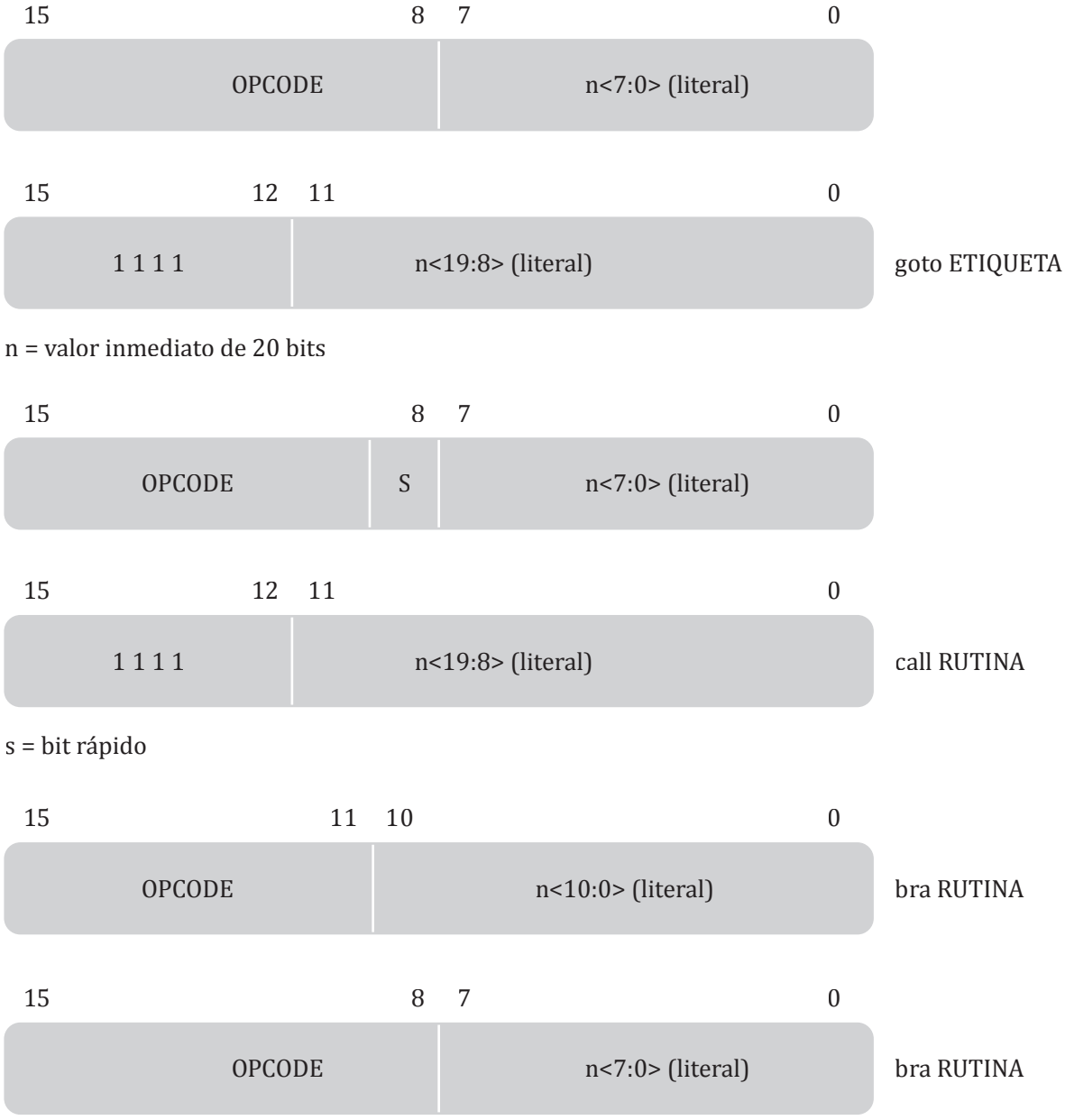
Mnemónico		Descripción
ADDLW	k	Sumar el valor literal k con el registro W
ANDLW	k	Operación AND entre el registro W y el valor literal k
IORLW	k	Operación OR entre el registro W y el valor literal k
LFSR	f,k	Mover el valor literal k de 12 bits a FSR(f)
MOVLB	k	Mover el valor literal k a BSR
MOVLW	k	Mover el literal k al registro W
MULLW	k	Multiplicar el valor literal k con el registro W
RETLW	k	Retornar con el valor literal k en el registro W
SUBLW	k	Restar el registro W del valor literal k
XORLW	k	OR Exclusiva entre el valor literal k y el registro W

Elaboración propia.

Instrucciones para operaciones de control

El formato para esta categoría de instrucciones se muestra en el gráfico 2.23. Aquí se puede observar hasta cuatro formatos diferentes, dependiendo del tipo de instrucción de control. Los dos primeros formatos toman dos ciclos de instrucción, ya que se codifican en 32 bits (ocupan dos direcciones de la memoria de programa). La notación n<7:0> establece los bits del 0 al 7 del número n, y la notación n<19:8>, los bits del 8 al 19. El número n representa la dirección absoluta que será cargada en el registro PC. En total son 20 bits, ya que el bit LSB del PC siempre es '0'. En el gráfico 2.24 se observan las instrucciones de control usadas en el lenguaje ensamblador del PIC18.

Gráfico 2.23. Formato de código máquina para las instrucciones de control



Elaboración propia.

Gráfico 2.24. Instrucciones de control

Mnemónico	Descripción
BC n	Salta a la dirección n si Carry = '1' lógico
BN n	Salta a la dirección n si el bit N = '1' lógico
BNC n	Salta a la dirección n si el bit Carry = '0' lógico
BNN n	Salta a la dirección n se el bit N = '0' lógico
BNOV n	Salta a la dirección n si no ha ocurrido un Overflow
BNZ n	Salta a la dirección n si el bit Z = '0' lógico
BOV n	Salta a la dirección n si ha ocurrido un *Overflow*
BRA n	Salta a la dirección n
BZ n	Salta a la dirección n si el bit Z = '1' lógico
CALL n,s	Salta a la dirección n. El PC se guarda en la pila. S indica que también se almacena en una pila el registro W, el STATUS y BSR
CLRWDT	Inicializar el Watch Dog Timer
DAW	Ajuste decimal del registro W
GOTO n	Saltar a la dirección n
NOP	Ninguna operación
POP	Retornar el último valor de la pila
PUSH	Guardar el PC en el último valor de la pila
RCALL n	Salto relativo a n
RESET	Generar un Reset por *software* del microcontrolador
RETFIE s	Retorno de la rutina de interrupción
RETLW k	Retorno de un rutina con el valor literal k en el registro W
RETURN s	Retorno de la subrutina. S indica si se rescatan de la pila el registro W, el STATUS y BSR
SLEEP	Pone al microcontrolador en modo de bajo consumo

Elaboración propia.

2.4 Las interfaces de entrada y salida (E/S)

En el mundo de los sistemas embebidos, las interfaces de entrada y salida se convierten en los puntos de acceso a la comunicación del procesador con dispositivos externos. Por ejemplo, para un diseño electrónico específico, un microcontrolador requiere comunicarse con un teclado matricial, un módulo LCD alfanumérico, un motor de pasos, un computador a través del puerto RS232 y un reloj en tiempo real (RTC).

Los microcontroladores se caracterizan por poseer pines de comunicación, conocidos como «puertos de entrada y salida (E/S)». Por ejemplo, el PIC18F4550 posee 40 pines en su empaque tipo DIP40, de los cuales 35 corresponden a E/S. Cada uno de estos pines tiene la característica de poder funcionar en tres estados: entrada digital, salida digital o como función especial de un periférico específico.

En el gráfico 2.25 se observa un esquema de un microcontrolador con dos pines de comunicación. El pin superior está configurado como entrada digital y se encuentra conectado a un pulsador y a una resistencia de *pull up*. Dado que el pin, al estar configurado como entrada, presenta alta impedan-

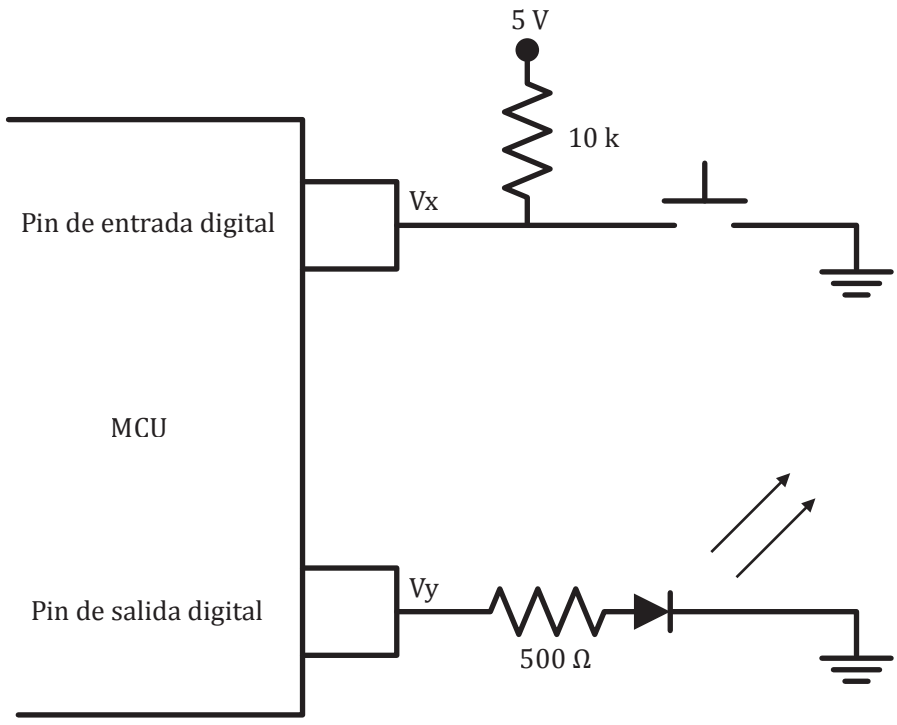
cia, se fijarán dos estados lógicos en el nivel de voltaje Vx en la entrada del pin: '0'(0 V) si el pulsador se encuentra pulsado o '1' (5 V) si el pulsador está abierto. El estado lógico del pin puede ser detectado por un programa que esté siendo ejecutado en el microcontrolador, de manera que el sistema embebido pueda interactuar con la pulsación.

Por otro lado, el pin inferior del gráfico 2.25 está configurado como salida digital y se encuentra conectado a una resistencia y al ánodo de un LED en serie. Si la lógica del programa establece la tensión de Vy a un nivel '0', el LED permanecerá apagado. Si, en cambio, Vy se establece en '1', el LED se encontrará encendido. En este caso, el pin de salida actuará como fuente de voltaje y generará una corriente capaz de encender el LED. Se debe tener mucho cuidado de que esa corriente no exceda los 25 mA, ya que este es el máximo valor que puede generar cada pin de un PIC18.

El acceso a los pines de E/S del microcontrolador normalmente se hace por grupos, denominados «puertos». Un puerto es un conjunto de pines que pueden ser escritos o leídos al mismo tiempo en un solo ciclo de instrucción. Por ejemplo, en el PIC18F4550 se cuenta con 5 puertos: Puerto A, Puerto B, Puerto C, Puerto D y Puerto E. Cada puerto contiene un número de pines diferente. Generalmente estos están asociados por la función de los periféricos. En el gráfico 2.26 se muestran dos puertos del PIC18F4550 y la numeración de sus pines. Usualmente no existe una regla que indique que los pines de los puertos deben tener una numeración consecutiva ascendente.

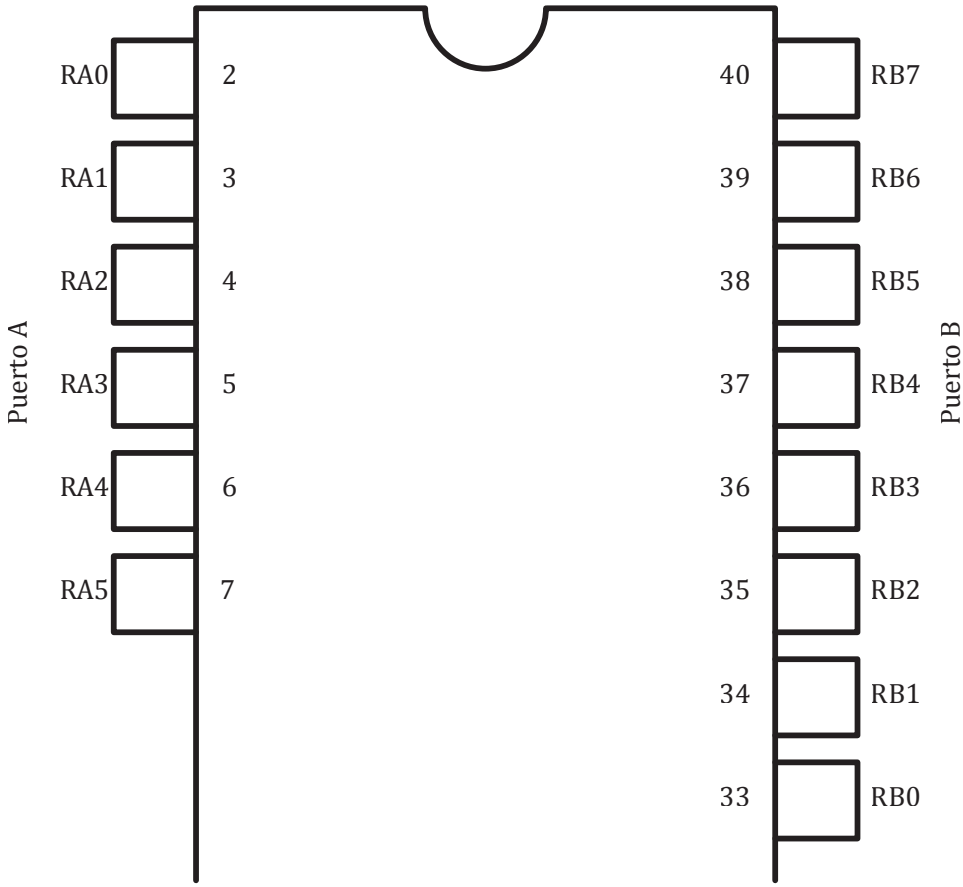
Para modificar o leer el estado de un puerto se debe acceder a su registro específico en el RFE. Por ejemplo, el Puerto B del PIC18F4550 está asociado a la dirección 0xF81 de la memoria de datos. Simplemente leyendo el registro apuntado por esta dirección se obtiene el valor del Puerto B, si es que sus pines han sido configurados como entrada digital. En el caso de que se configure el Puerto B como salida digital, la escritura de un valor en esta dirección establecerá el nivel de voltaje en los pines del Puerto B.

Gráfico 2.25. Comunicación de los pines de E/S con elementos de entrada y de salida



Elaboración propia.

Gráfico 2.26. Puertos A y B del encapsulado del PIC18F4550



Elaboración propia.

Para configurar un puerto como entrada o salida se hace uso de los registros TRISx, donde x puede ser A, B, C, D o E (para el caso específico del PIC18F4550). El nombre TRIS viene de Tri-estado, lo que significa que cada pin puede tener tres estados: entrada digital, salida digital o un estado asociado a la función de su periférico. Los registros TRISx también se encuentran como RFE con una dirección específica para cada puerto (por ejemplo, el registro TRISA tiene la dirección 0xF92 y TRISB la dirección 0xF93 en el mapa de memoria de datos del PIC18F4550). La posición del bit en el registro TRISx especifica el número de pin del puerto que será configurado como entrada o salida. Un '0' en una posición del registro TRISx (donde la posición puede variar de 0 a 7) implica que el pin será salida digital en esa posición, y un '1', que operará como entrada digital. Por defecto, todos estos pines están configurados como entradas digitales. Es posible combinar estados de entrada y salida entre los pines de un mismo puerto. En el gráfico 2.27 se muestra la configuración del Puerto B a través del registro TRISB.

Cuando un pin de un puerto es configurado para trabajar bajo el control de su periférico respectivo, el contenido del registro TRISx puede o no interesar, dependiendo del tipo de periférico. Para esto, se debe revisar en la hoja de datos del microcontrolador la influencia del registro TRISx sobre un periférico específico.

En algunas situaciones, con versiones de las familias PIC16 y PIC12, se encontraron ciertos problemas en los estados de los pines del puerto cuando se realizaban cambios de estado a una velocidad muy rápida, teniendo cargas que exigían corrientes cercanas a los 25 mA. Los valores de voltaje generados no aparecían en el momento debido y, como resultado, se detectaba un '0' cuando se debía obtener

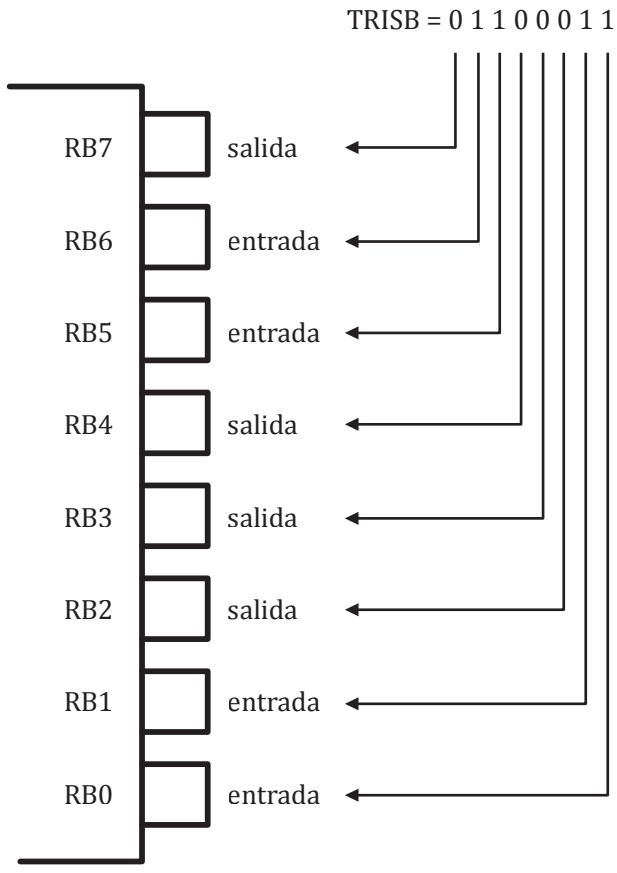
un '1'. Para evitar este problema, la familia PIC18 viene con un nuevo conjunto de registros *latch* de nombre LATx (donde x puede ser A, B, C, D y E para el PIC18), que también se acceden como RFE. Cada registro referencia a un *latch* conectado a la lógica del puerto y permite generar un estado, que aparecerá en el pin un ciclo de instrucción después de cargado el valor en el registro, estabilizando el estado lógico del puerto. En el gráfico 2.28 se muestra un diagrama genérico de la lógica digital detrás de cada pin de un puerto en el microcontrolador PIC18.

2.5 El Reset

Un Reset es un evento que ubica al microcontrolador en un estado inicial. Este estado inicial indica que el programa volverá a ser ejecutado desde el principio. Ante esta condición, ocurrirán los siguientes eventos:

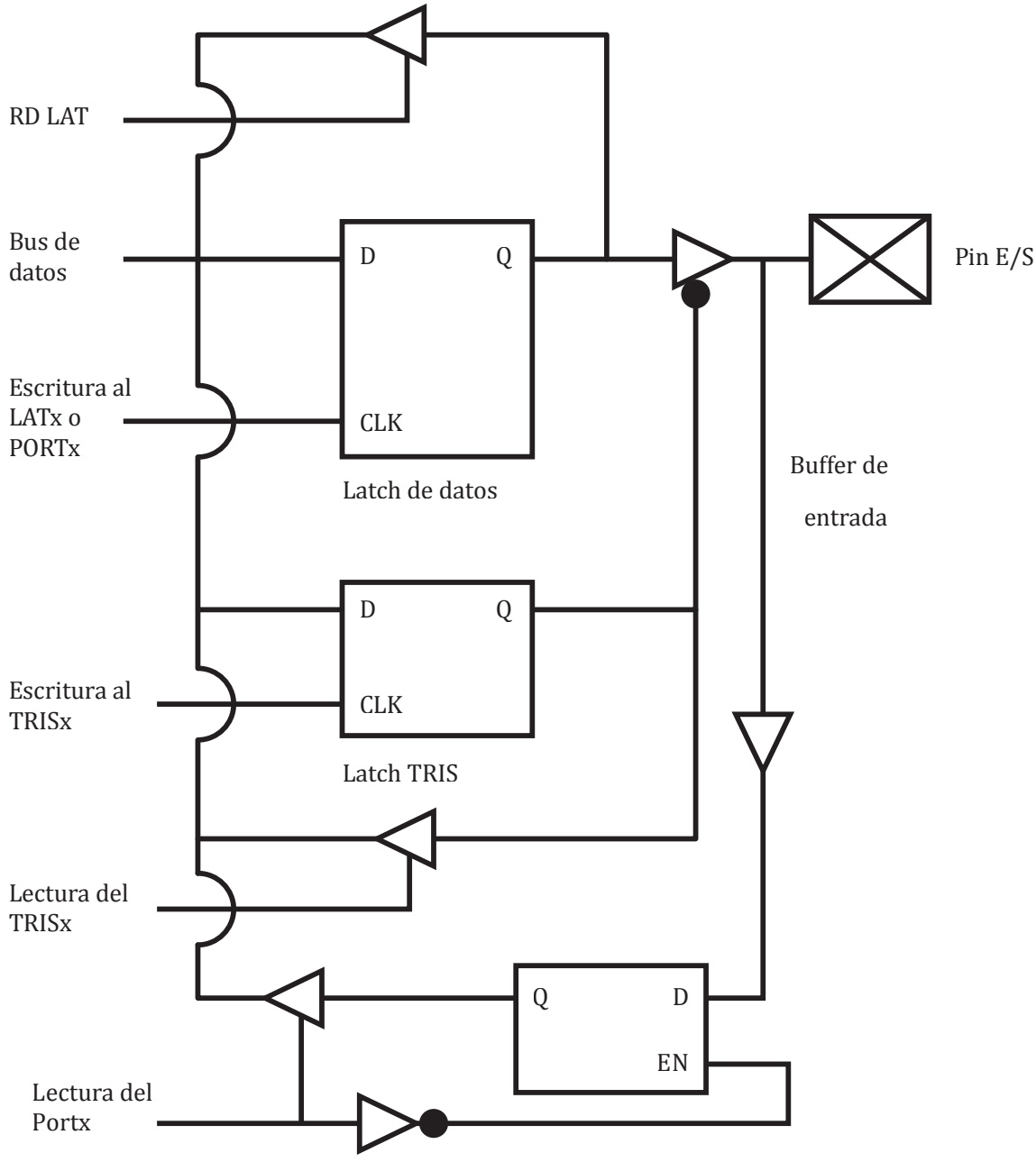
1. El puntero de pila STKPTR apuntará a la primera dirección válida.
2. Los registros RFE pasarán a su estado inicial.
3. Todos los puertos E/S se configuran como entradas digitales.
4. Los temporizadores se inicializan en 0x00.
5. Los modos de bajo consumo quedan desactivados.
6. El PC apunta a la primera dirección 0x0000 de la memoria FLASH.
7. El reloj del Perro guardián se inicializa en 0x00.

Gráfico 2.27. Ejemplo de configuración del Puerto B con el registro TRISB



Elaboración propia.

Gráfico 2.28. Lógica digital detrás de cada pin de E/S



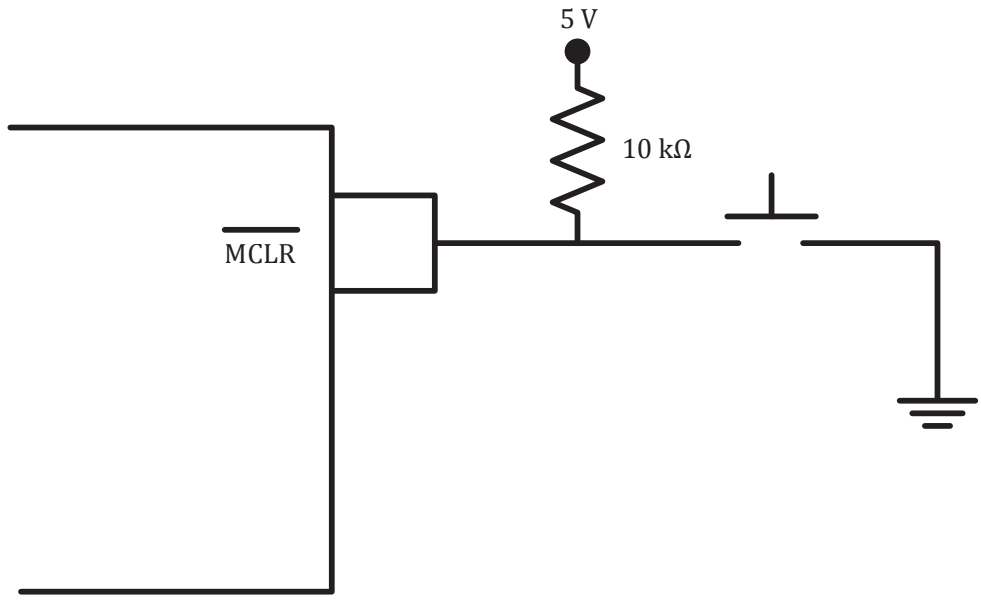
Elaboración propia.

Existen muchos modos para generar un Reset en el microcontrolador. El modo más conocido utiliza al pin MCLR, un pulsador y una resistencia de *pull up*, tal como se muestra en el gráfico 2.29. Cuando el estado lógico del pin es '1', el programa seguirá su ejecución normal. Si el pin se pone a '0', el microcontrolador entra en estado de Reset y se mantendrá allí hasta que el voltaje se reponga a '1'. Este modo es el único que permite generar un Reset manual (a través de la pulsación de un pulsador), ya que los otros modos dependen de estados internos del microcontrolador. Los PIC18F2455/2550/4455/4550 poseen varios tipos de modos de Reset:

a. Power on Reset (POR o encendido de la fuente de voltaje del microcontrolador).
b. MCLR Reset, durante operación normal.
c. MCLR Reset durante operación en modo de manejo de energía.
d. Reset por desbordamiento del contador del Perro guardián.
e. Brown Out Reset Programable (BOR o por baja de tensión de alimentación por debajo de un umbral).
f. Instrucción de Reset.
g. Reset por pila llena.
h. Reset por pila vacía.

Los eventos de Reset del microcontrolador se pueden detectar a través del registro RCON. Los 5 bits menos significativos del registro indican que un evento de Reset específico ha ocurrido. En la mayoría de los casos, estos bits solo pueden ser borrados por el evento y deben ser puestos a '1' por la aplicación después de ocurrido el evento. El estado de estas banderas se puede leer para conocer el tipo de Reset que acabó de ocurrir y, a partir de esto, tomar una acción diferente en el programa. En el gráfico 2.30 se muestra una imagen del registro RCON y la función de sus bits.

Gráfico 2.29. Reset externo a través del pin MCLR



Elaboración propia.

Gráfico 2.30. Registro RCON y sus bits de estado

IPEN	SBOREN	——	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$

IPEN: Habilitador de la prioridad de la interrupción
1: Niveles de prioridad habilitados.
0: Compatibilidad en las interrupciones con el PIC16

SBOREN: Habilitador del módulo de detección de caída de energía
1: BOR activo
0: BOR desactivado

$\overline{RI}$: Bandera indicadora de la ejecución de la instrucción RESET
1: No se ejecutó el RESET
0: Se ejecutó el comando RESET

$\overline{TO}$: Bandera de RESET por desbordamiento del Perro guardián
1: No se produjo el evento
0: Se desbordó el Perro guardián

$\overline{PD}$: Bandera de detección del modo de bajo consumo
1: No se generó el modo de bajo consumo
0: Se ejecutó la instrucción SLEEP

$\overline{POR}$: RESET por activación de la fuente de energía
1: No ocurrió el evento
0: RESET por activación de la fuente de energía

$\overline{BOR}$: RESET por baja de tensión
1: No ocurrió el evento
0: Se generó un evento de baja tensión

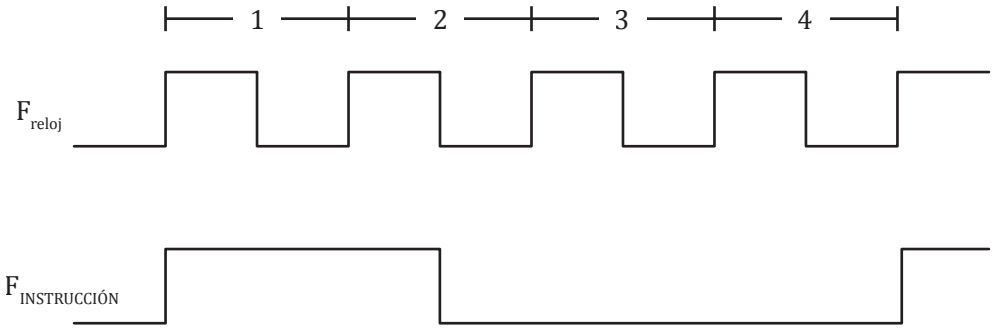
Elaboración propia.

2.6 Temporización y opciones de reloj

Todo microcontrolador requiere de un elemento capaz de generar una señal de reloj a determinada frecuencia, que marque el ritmo de ejecución de instrucciones del procesador. La señal de reloj puede ser obtenida de diversas formas: con un oscilador externo, un cristal externo, un circuito RC o un oscilador interno. Como se puede observar del gráfico 2.31, la frecuencia de ejecución de instrucciones demora cuatro ciclos de reloj del oscilador, con lo cual, mientras mayor sea la frecuencia de reloj, mayor será la velocidad de ejecución de instrucciones.

Si, por ejemplo, la frecuencia de reloj se fijara en 20 MHz, entonces la frecuencia de instrucciones sería 20÷4 = 5 MHz. Esto implica que cada instrucción será ejecutada en un tiempo equivalente a $\frac{1}{5MHz} = 0.2\mu s$. Si se asume que todas las instrucciones ejecutadas por el procesador en un segundo consumen un solo ciclo de instrucción, se puede decir que este tiene una velocidad de 5 MIPS.

Gráfico 2.31. Relación entre la frecuencia de reloj y la frecuencia de ejecución de instrucciones



Elaboración propia.

Los microcontroladores PIC18F2455/2550/4455/4550 presentan un oscilador y un sistema de reloj diferente que las familias PIC18F antecesoras. El hecho de que posean un módulo USB interno genera el requerimiento de un reloj muy estable, lo cual hace necesaria una fuente de reloj independiente y de un valor determinado en 48 y 6 MHz, para que puedan ser compatibles con las especificaciones de los estándares USB Low-Speed y el USB Full-Speed.

Para acomodar estos requerimientos, los PIC18F2455/2550/4455/4550 incluyen un nuevo modo de proveer un reloj de 48 MHz para la operación Full-Speed. Como este reloj depende del cristal externo (el cual es la fuente típica de generación de frecuencia usada en los microcontroladores), se ha visto necesario incorporar un sistema adicional de Postcalers y Prescalers (divisores de frecuencia) para acomodar un alto rango de frecuencias de oscilación externas a los requerimientos de 48 MHz para el bus USB.

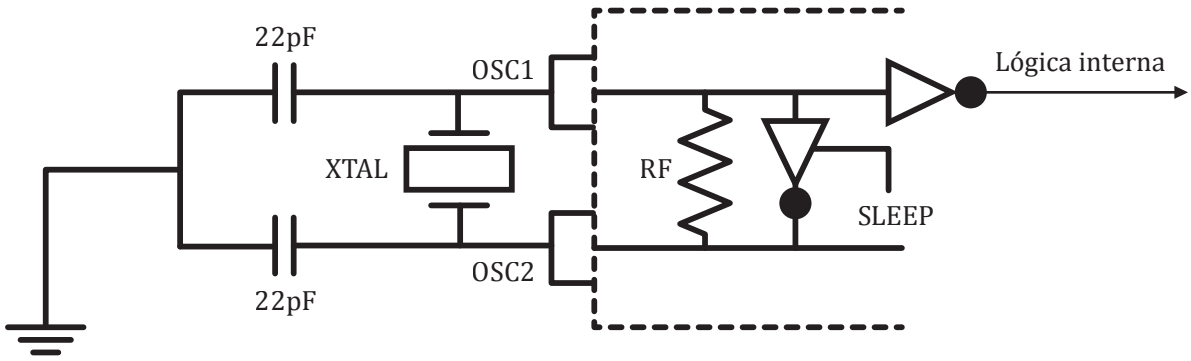
El PIC18F4550 puede ser operado en 12 tipos distintos de modos de oscilación. Cuatro de estos modos involucran el uso de dos tipos de oscilador a la vez. Estos modos son los siguientes:

1. XT Cristal/Resonador.
2. XTPLL Cristal/Resonador con PLL activo.
3. HS Cristal/Resonador de alta velocidad.
4. HSPLL Cristal/Resonador de alta velocidad con PLL activo.
5. EC Reloj externo con salida $F_{RELOJ}/4$ en la línea RA6.
6. ECIO Reloj externo con RA6 configurado como E/S.
7. ECPLL Reloj externo con PLL habilitado y salida $F_{RELOJ}/4$ en RA6.
8. ECPIO Reloj externo con PLL habilitado, RA6 configurado como E/S.
9. INTHS Oscilador interno usado como fuente de reloj del microcontrolador, oscilador HS usado para la fuente de reloj del USB.
10. INTXT Oscilador interno usado como fuente de reloj del microcontrolador, oscilador XT para la fuente de reloj del USB.
11. INTIO Oscilador interno usado como fuente de reloj del microcontrolador, oscilador EC usado como fuente de reloj del USB y RA6 configurado como puerto E/S.
12. INTCK0 Oscilador interno usado como fuente de reloj del microcontrolador, oscilador EC usado como fuente de reloj del USB y salida $F_{RELOJ}/4$ para RA6.

Debido a los requerimientos especiales del módulo USB, el diseño de los PIC18F2455/2550/4455/4550 contiene una nueva propuesta para el módulo de operación de reloj. En los dispositivos previos, la señal de reloj para el CPU y los periféricos partía de una sola fuente de oscilación. Las fuentes típicas eran la primaria, la secundaria y el oscilador interno. Con el PIC18F4550, el oscilador primario se vuelve parte del módulo USB, que no puede ser asociado a ninguna otra fuente de reloj. Así, el módulo USB debe usar un reloj de la fuente primaria. Sin embargo, el CPU del microcontrolador y sus periféricos pueden usar relojes separados, conectados a la fuente secundaria o del oscilador interno.

En los modos HS, HSPLL, XT y XTPLL, un cristal o resonador cerámico debe ser conectado entre los pines OSC1 y OSC2 para establecer la oscilación. El gráfico 2.32 muestra las conexiones del resonador a estos pines. En modo XT el cristal o resonador a utilizar debe ser de 4 MHz. Para el modo HS el cristal o resonador puede tomar los valores de 4, 8 y 20 MHz.

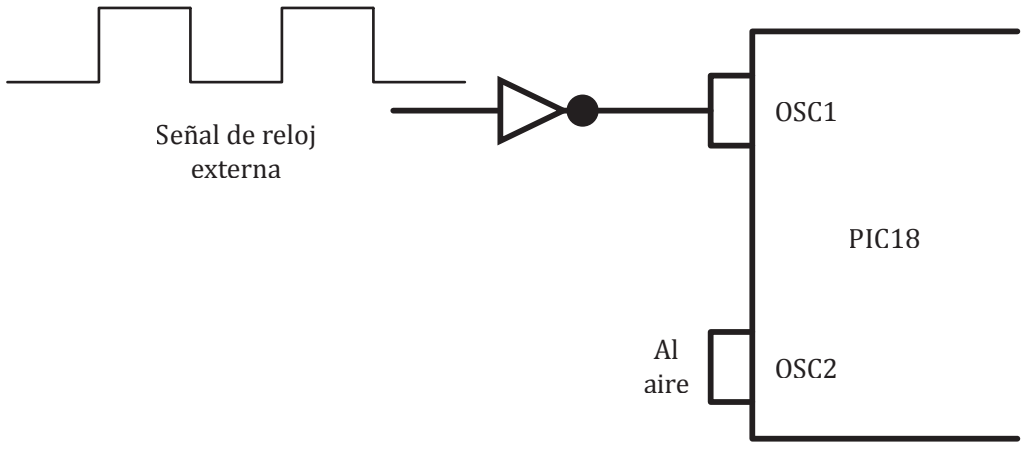
Gráfico 2.32. Configuración del cristal externo para las operaciones XT, HS o HSPLL



Elaboración propia.

Un *postcaler* interno permite al usuario seleccionar una frecuencia de reloj menor a la del cristal o resonador. La división de frecuencia es determinada por los bits de configuración CPUDIV. Los usuarios pueden seleccionar una frecuencia de reloj de 1/2, 1/3 o 1/4 de la frecuencia del oscilador. En modo HS se puede usar una señal de reloj externo. En este caso, el pin OSC2 se puede dejar al aire, tal como se muestra en el gráfico 2.33.

Gráfico 2.33. Conexión de la señal de reloj externo en modo HS

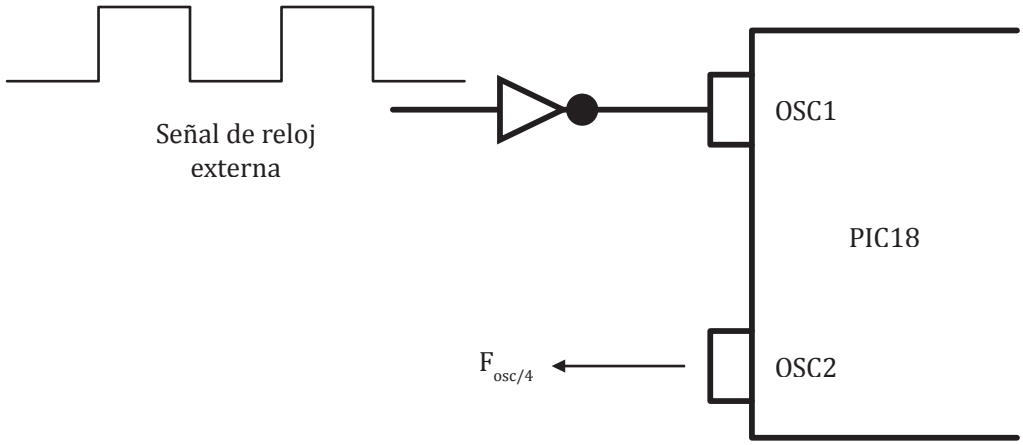


Elaboración propia.

En los modos EC, ECIO, ECPLL y ECPIO se requieren fuentes de reloj externas para que sean conectadas al pin OSC1. No existen requerimientos de tiempo de inicio al salir de un Reset de energía (Power-on Reset) o después de salir del modo *sleep*.

En el modo EC y ECPLL, la frecuencia del oscilador se divide entre 4 y se encuentra disponible en el pin OSC2. Esta señal puede ser usada para propósitos de test o para una lógica de sincronización. El gráfico 2.34 muestra las conexiones de los pines para el modo EC.

Gráfico 2.34. Conexión del reloj externo en los modos EC y ECPLL



Elaboración propia.

Los modos ECIO y ECPIO del oscilador funcionan de forma muy similar a los modos EC y ECPLL, con la excepción de que el pin OSC2 se vuelve un pin de propósito general. El pin E/S se convierte ahora en el pin RA6 del Puerto A.

El PIC18F4550 incluye un circuito interno denominado Phase Locked Loop (PLL). Este circuito permite elevar la frecuencia de entrada por un factor fijo. Es así que el PLL se encuentra especialmente diseñado para las aplicaciones USB con osciladores de baja velocidad, y también puede ser usado para las fuentes de reloj del microcontrolador.

El PLL es habilitado en los modos de oscilador HSPLL, XTPLL, ECPLL y en modos ECPIO. El PLL está diseñado para producir una frecuencia fija de 96 MHz como reloj de referencia para una entrada externa de 4 MHz. Es decir, presenta un factor de multiplicación de 24. La salida es dividida y usada para el USB y el CPU. Debido a que el PLL tiene una frecuencia fija de entrada y salida, existen 8 opciones de *prescaler* para dividir la frecuencia del reloj externo y convertirla a 4 MHz, de manera que la frecuencia de salida sea los 96 MHz del PLL.

Adicionalmente, existe un *postcaler* separado para obtener la frecuencia del microcontrolador desde el PLL. Esto permite que el USB y el microcontrolador usen la misma fuente de oscilación, pero con diferentes frecuencias de reloj. A diferencia de los modos de *Postcaler* para XT, HS y EC, los divisores disponibles son 1/2, 1/3, 1/4 y 1/6 de la salida PLL.

Los modos HSPLL, ECPLL y ECPIO hacen uso del oscilador en modo HS para frecuencias de hasta de 48 MHz. El *prescaler* divide la entrada del oscilador hasta por un factor de 12 para producir los 4 MHz que requiere el PLL. El modo XTPLL solo puede utilizar una frecuencia de entrada de 4 MHz, la cual se conecta directamente al PLL.

El PIC18F4550 incluye un oscilador interno capaz de generar dos señales de reloj diferentes. Cualquiera puede ser usada como fuente de reloj para el microcontrolador. Si el módulo USB no va a ser usado, se puede utilizar el oscilador interno e ignorar la señal de reloj externo en los pines OSC1 y OSC2.

La salida principal (INTOSC) es una señal de reloj de 8 MHz, que puede ser usada directamente como señal de reloj del sistema. Esta también se conecta a un *Postcaler* de nombre INTOSC, el cual puede proveer un rango de frecuencias de reloj desde 31 kHz hasta 4 MHz. La salida INTOSC es habilitada cuando se selecciona un reloj de frecuencia entre 125 kHz y 8 MHz.

Adicionalmente, el PIC18F4550 ofrece un oscilador secundario para el *Timer* 1. Este oscilador, en todos los modos de manejo de energía, es frecuentemente usado para funciones de reloj de tiempo real. Normalmente, se utiliza un cristal de 32,768 kHz conectado a los pines RC0 y RC1.

2.7 Modos de ahorro de energía

El manejo de energía es un proceso fundamental en un microcontrolador. En muchas aplicaciones se requiere únicamente realizar operaciones bajo ciertos eventos externos, que pueden ocurrir en instantes de tiempo muy poco frecuentes, por lo cual la mayor parte del tiempo el procesador se encuentra en espera. Este esquema se complica cuando la fuente de alimentación está formada por baterías que tienen un tiempo de vida limitado. Para esto, el PIC18 ofrece una serie de opciones que permiten reducir el consumo de corriente por parte del microcontrolador. Claro está que este modo de bajo consumo podrá ser activado mientras el procesador se encuentre en espera de un evento externo y no ejecute instrucciones de forma continua.

Los PIC18F2455/2550/4555/4550 ofrecen un total de siete modos de operación para un manejo eficiente de la energía. Estos modos proveen una variedad de opciones para seleccionar la forma de conservación de energía en diversas aplicaciones, donde los recursos pueden estar limitados.

Existen tres categorías de modos de manejo de energía:

- Modo RUN
- Modo IDLE
- Modo SLEEP

Estas categorías definen cuáles porciones del microcontrolador se encuentran conectadas a la señal de reloj y a qué velocidad máxima pueden operar. Los modos RUN o IDLE pueden utilizar cualquiera de las tres fuentes de reloj (primaria, secundaria y reloj interno). El modo *sleep* no utiliza una fuente de reloj.

Los modos de manejo de energía incluyen diversas características de ahorro energético ofrecidas en las versiones previas del PIC18. Una de estas es la característica de cambio de reloj, permitiendo al controlador usar el oscilador del *Timer* 1 en lugar del oscilador primario. También se incluye el modo *sleep*, ofrecido por todos los PIC, donde los relojes del microcontrolador se encuentran detenidos.

Modo RUN

En el modo RUN los relojes, tanto para el CPU como para los periféricos, se encuentran activos. La diferencia entre estos modos es la fuente de reloj.

El modo PRI_RUN es el modo normal de operación, ejecutando las instrucciones del microcontrolador a la máxima energía. Este también es el modo por defecto luego del Reset del dispositivo.

El modo SEC_RUN es el modo compatible con la característica de cambio de reloj ofrecida por los PIC18. En este modo, el CPU y sus periféricos son conectados al reloj oscilador del *Timer* 1. Esto da a los usuarios la opción de reducir el consumo de energía mientras se utiliza un reloj de alta precisión como fuente de reloj.

En el modo RC_RUN, el CPU y los periféricos se encuentran conectados al bloque de oscilador interno a través del multiplexor INTOSC. La fuente de reloj primaria se apaga. Cuando se utiliza la fuente de reloj INTRC, se provee la mejor conservación de energía para todos los modos RUN mientras aún se ejecuta código. Este modo funciona bien para aplicaciones que no requieren relojes de alta velocidad.

Modo SLEEP

El modo *sleep* en el PIC18F4550 es similar al modo ofrecido en las versiones PIC18F anteriores. Ejecutando la instrucción *sleep* se apaga el oscilador seleccionado. Todos los bits de estado de fuente de reloj se cargan con '0'.

Ingresar al modo *sleep* es posible desde cualquier otro modo y no requiere un cambio de reloj. Esto se debe a que no se requieren fuentes de reloj para controlar el modo *sleep*. Si el Perro guardián o *Watch Dog Timer* (WDT) se encuentra seleccionado, la fuente de señal RC interna continuará operando. Si el oscilador del *Timer* 1 se encontrara habilitado, también continuará operando.

Cuando un evento de *wake up* (salida del modo *sleep*) ocurre durante el modo *sleep* (debido a cualquier interrupción, un Reset o el *time-out* del WDT), el dispositivo no tendrá un reloj hasta que los bits SCS1:SCS0 del registro OSCCON se vuelvan disponibles.

Modo IDLE

El modo IDLE permite que el CPU se mantenga selectivamente desactivado mientras que los periféricos se mantienen operando. La selección de un modo particular IDLE permite al usuario gestionar el consumo de energía.

Si el bit IDLEN está en '1' cuando la instrucción *sleep* se ejecuta, los periféricos estarán conectados a la fuente de reloj seleccionada por los bits SCS1:SCS0. Sin embargo, el CPU no tendrá reloj. Los bits de selección de reloj no estarán afectados. Poniendo el bit IDLEN a '1' y ejecutando la instrucción *sleep* se obtendrá un método rápido para cambiar de un modo RUN a un modo IDLE.

Si se selecciona el WDT, la fuente de reloj RC interna continuará operando. Si el oscilador del *Timer* 1 se habilita, también continuará operando.

Dado que el CPU no ejecuta instrucciones, las únicas formas de salir del modo IDLE son a través de interrupciones, por el Reset del WDT o por un Reset externo. Cuando un evento de despertar ocurre, la ejecución del CPU se retrasa un intervalo de tiempo mientras se vuelve disponible para ejecutar el código. Cuando el CPU comienza a ejecutar el código, utilizará el mismo reloj del modo IDLE seleccionado. Por ejemplo, cuando se despierta del modo RC_IDLE, la fuente de oscilador del CPU y los periféricos será el oscilador interno.

Para salir del modo *sleep* o modo IDLE se requiere una interrupción, un *time-out* del WDT o un Reset.

2.8 El Perro guardián

El Perro guardián o *Watch Dog Timer* (WDT) es un circuito de conteo que utiliza un reloj interno. Posee la opción de generar un evento de Reset al microcontrolador si llega hasta cierto valor de cuenta. Este sistema es muy útil cuando se requiere garantizar que el microcontrolador continuará funcionando a pesar de que se puedan producir errores de lectura en memoria (error de *software*) que lleven al procesador a un bucle indefinido del que no pueda salir a menos que sea reinicializado.

Para los PIC18F2455/2550/4455/4550, el WDT utiliza la señal de reloj interna producida por el circuito RC. Cuando se habilita el WDT, también ocurre lo mismo con el oscilador interno. El periodo nominal de la señal de reloj del WDT es de 4 milisegundos.

Los 4 milisegundos de periodo del WDT se multiplican por un *postscaler* de 16 bits. La salida del *postscaler* es seleccionada de un multiplexor controlado por los bits de configuración de la memoria de programa en el Registro 2H. Los periodos disponibles varían entre 4 milisegundos hasta 131.072 segundos. El WDT y el *postscaler* pueden ser inicializados mediante dos eventos: la instrucción SLEEP o la instrucción *clrwdt*.

Preguntas de repaso

1. Defina el concepto de computador.
2. ¿Cuáles son las principales diferencias entre la arquitectura del procesador Von Neumann y la arquitectura Harvard?
3. Mencione las partes que componen a la unidad central de proceso.
4. Si la frecuencia de reloj externa de un PIC18F es de 20 MHz, ¿cuánto es el tiempo que demora en ejecutar una simple instrucción?
5. Explique por qué las instrucciones del PIC18F ocupan siempre las direcciones pares de la memoria de programa.
6. ¿Cuál es la función de la pila del microcontrolador PIC18?
7. ¿Cuál es el mayor resultado que se puede obtener de la unidad de multiplicación del PIC18?
8. Si en el registro STATUS el bit Z se encuentra en estado '1', ¿qué puede inferir de esta condición?
9. ¿Cuál es el procedimiento para acceder al contenido de un registro en la dirección 0x07AE de una memoria SRAM?
10. Explique para qué se divide la memoria SRAM del PIC18 en bancos.
11. ¿En qué consiste el modo de direccionamiento indirecto? ¿Qué registros se encuentran involucrados en este proceso?
12. ¿Cuál es la diferencia entre el registro PCLATU y el registro PCU? ¿Contienen la misma información?
13. ¿Qué función cumple el registro TBLPTR en el acceso a la memoria de programa? ¿Es similar al registro TABLAT o existen diferencias?
14. Explique cuál es la función del registro puntero de pila STKPTR.
15. Indique dos diferencias entre una memoria EEPROM y una memoria de tipo FLASH.
16. ¿En qué se diferencia un mnemónico de un código de operación (*Opcode*)?

17. Si los 4 pines más significativos del Puerto B son configurados como salidas digitales y los 4 bits menos significativos se configuran como entradas digitales, ¿cuál es el valor binario que deberá cargarse en el registro TRISB para obtener esta configuración?
18. Explique qué es un evento de Reset y cuáles son las formas en que este evento se puede generar en el microcontrolador PIC18.
19. ¿Cuál es la función que cumple un PLL interno en un microcontrolador y qué ventajas presenta el uso de este circuito para las aplicaciones con el sistema embebido?
20. ¿Cuántos modos de ahorro de energía posee el PIC18?
21. ¿Cuál es la función del Perro guardián (*Watch Dog Timer*) en el PIC18?

Respuestas

Respuestas del capítulo 2:

http://bit.ly/1KCdNXw

Capítulo 3. El compilador para PIC18F. El MPLAB X IDE

Para poder comenzar con la programación de los microcontroladores PIC18F, es necesaria una herramienta de *software* que permita redactar el código (ya sea en lenguaje ensamblador o ANSI C), verificar que la sintaxis del programa esté correctamente definida, convertir la secuencia de instrucciones al código máquina, generar los archivos de salida correspondientes y, finalmente, permitir descargar el archivo de programa en la memoria FLASH del *Target* a través del programador.

Existen diversas herramientas de *software* para implementar todo el proceso de flujo de desarrollo que permite la programación de microcontroladores de 8 bits de CPU de la marca Microchip. Sin embargo, la herramienta más popular es el *software* MPLAB X IDE, otorgado por el mismo fabricante para que sea descargado desde su página web.

En este capítulo se presentará el *software* MPLAB X IDE y sus características más importantes, y se explicará cómo crear un nuevo proyecto. Se revisará, además, la metodología para simular un programa (y así evitar utilizar siempre el *Target* para verificar el funcionamiento) y, finalmente, se mostrará el compilador MPLAB XC8, que permitirá la generación de código en lenguaje ANSI C, que es muy popular en estos tiempos y facilita enormemente el desarrollo de programas de alta complejidad. Este lenguaje será presentado en detalle en el octavo capítulo del libro.

3.1 IDE

Las siglas del idioma inglés IDE (*Integrated Development Enviroment* o ambiente de desarrollo integrado) hacen referencia a una plataforma integrada de desarrollo que permite realizar todo el flujo de programación de los microcontroladores haciendo uso de dos lenguajes: ensamblador o ANSI C. El MPLAB X IDE es una herramienta entregada gratuitamente por Microchip para que los usuarios puedan implementar cualquier tipo de aplicación dentro de los microcontroladores.

El MPLAB X IDE es un *software* desarrollado en IDE abierto NetBeans de Oracle y se encuentra diseñado para operar bajo sistemas operativos Windows (actualmente bajo Windows 7 y Windows 8), Linux y MAC OS. El programa permite la redacción de código, compilación, ensamblado, simulación, generación del archivo ejecutable y grabación del microcontrolador. Las versiones de MPLAB X se actualizan continuamente en la página web de Microchip. Hasta la edición de este libro, la última versión de MPLAB X IDE es la 2.20.

Adicionalmente, el *software* contiene librerías para facilitar la programación de todos los modelos de microcontroladores de la línea de Microchip. Cada librería incluye los nombres de los registros internos, la nomenclatura de sus bits, rutinas de inicialización, macros, etc. Además, el MPLAB X cuenta con herramientas adicionales que permiten crear código predefinido para facilitar el desarrollo de aplicaciones de mayor nivel de complejidad.

El entorno de desarrollo se compone de los siguientes elementos:

1. Un editor de texto que permite la redacción de código. Un proyecto puede tener relacionados varios archivos de texto con partes del programa.
2. El administrador de proyectos, que es una ventana donde se pueden asociar diversos archivos relacionados al mismo proyecto.
3. El ensamblador, que se encarga de generar varios archivos objeto o de salida del proyecto. Este trabaja muy de cerca con el compilador, que se encarga de verificar la sintaxis del programa y presentar la lista de errores, en caso de que los haya, enumerados línea por línea.
4. El enlazador, cuya función es unir todos los archivos objeto en uno solo, y así generar el archivo final de salida (de extensión .hex), que será grabado en el *Target*.

Adicionalmente, se requiere de un programador como el PICkit2 o PICkit3. Esta herramienta es necesaria para descargar el código máquina del programa en la memoria FLASH del sistema embebido.

3.2 Creación de un nuevo proyecto en MPLAB X usando el compilador MPASM

Para iniciar un nuevo proyecto con la herramienta de desarrollo, el primer paso consiste en abrir el programa MPLAB X IDE v2.20 desde el escritorio de Windows (para efectos de los ejemplos vistos, se hará uso de este sistema operativo). Posteriormente, aparecerá la ventana principal del programa, tal como se muestra en el gráfico 3.1. Aquí se puede observar un menú de opciones compuesto por 10 elementos: Archivo (*File*), Edición (*Edit*), Vista (*View*), Navegación (*Navigate*), Fuente (*Source*), Refractor (*Refractor*), Ejecución (*Run*), Depurador (*Debug*), Equipo (*Team*), Herramientas (*Tools*), Ventana (*Window*) y Ayuda (*Help*). Las opciones del menú cumplen con las siguientes funciones:

1. La opción *File* permite abrir, guardar, crear y adherir nuevos archivos al proyecto, además de guardar los cambios hechos en el proyecto, entre otras funciones adicionales.
2. La opción *Edit* o de edición contiene las opciones clásicas de copiar, pegar, cortar, deshacer, borrar, seleccionar, búsqueda de palabras y reemplazo de las mismas.
3. La opción *View* permite mostrar una serie de ventanas que son de utilidad para visualizar información del proyecto y del programa durante su simulación.
4. La opción *Navigate* posee opciones de búsquedas avanzadas de archivos, símbolos, líneas de código, entre otras, en el proyecto en curso.
5. La opción *Source* permite seleccionar bloques de código para mover el margen a la derecha o izquierda, duplicar el contenido o comentar el código, entre otras opciones que son altamente utilizadas por los programadores.
6. La opción *Refractor* se aplica para programas elaborados en lenguaje C y permite eliminar y renombrar variables y funciones, copiar librerías de un bloque de código a otro y deshacer los cambios.
7. La opción *Run* contiene una lista de funciones para compilar el código, modificar las opciones del proyecto como el cambio del modelo de microcontrolador, depurador y programador, permite la visualización de los bits de configuración del dispositivo, configurar algunas opciones del proyecto, y las herramientas de ensamblaje, enlace y depuración. Permite iniciar el proceso de simulación del programa realizado.

8. La opción *Debug* contiene los botones para el control de la simulación del programa. Aquí se establecen los *Breakpoints*, ejecución paso por paso, Reset, ventanas de visualización de variables, entre otros.
9. La opción *Team* es muy interesante y permite visualizar los últimos cambios realizados por hora y fecha al código, como para llevar un control de las modificaciones realizadas. Al mismo tiempo permite, ubicar problemas o generar reportes de inconvenientes detectados en el código.
10. La opción *Tools* permite verificar el estado de la licencia del IDE, comparar el contenido de los archivos, activar *Plugins* y modificar opciones del proyecto, como el tipo de compilador, el microcontrolador a utilizar, cambiar el tipo y tamaño de letra de la ventana de texto, entre muchas otras opciones de configuración.
11. La opción *Window* habilita ventanas para la visualización de los contenidos de las memorias de programa, SRAM, pila y EEPROM del PIC18F. Además, permite visualizar variables durante la simulación, invocar al simulador de reloj (*Stopwatch*), activar la ventana de salida, entre muchas otras opciones.
12. Finalmente, la opción *Help* contiene información de la versión del programa MPLAB X, permite realizar actualizaciones y contiene enlaces a tutoriales en internet para obtener ayuda sobre el manejo del *software*.

Gráfico 3.1. Ventana principal del *software* MPLAB X IDE

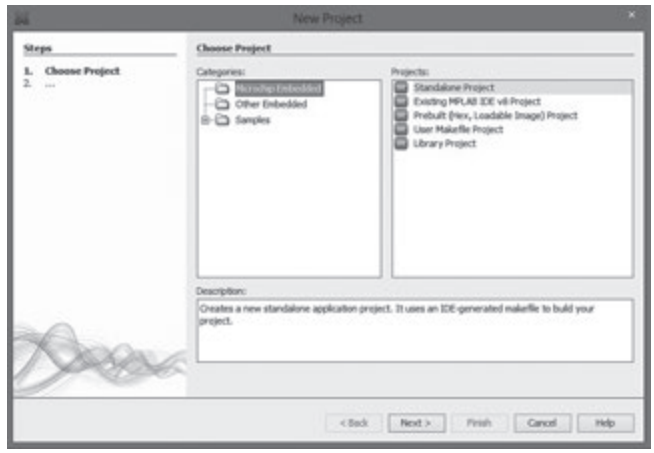
Elaboración propia.

Continuando con la creación del proyecto, el primer paso consiste en ingresar a la opción de menú *File->New Project*, que desplegará como resultado una ventana inicial que guiará al programador en configuración de un nuevo proyecto de manera simple y organizada. La ventana se muestra en el gráfico 3.2, donde se visualizan dos listas, una de categorías (*Category*) y otra de proyectos (*Projects*). La primera lista permite elegir entre crear un proyecto desde cero (en inglés se conoce como *stand alone project*) o un proyecto precompilado que involucra código definido que solo debe ser completado por el programador. En la segunda lista de proyectos se define el tipo de proyecto que se realizará, es decir, si se va a programar un código en el *firmware* de un microcontrolador, si se va a crear una librería o si se va a importar un proyecto desde el MPLAB IDE (IDE usado por Microchip por muchos años, que fue reemplazado por el MPLAB-X), entre otras opciones. Para editar un nuevo proyecto, se selecciona de la primera lista *Category* la opción *Microchip Embedded*, y de la lista *Projects*, la opción *Standalone Project*. Aquí se deberá hacer clic sobre el botón *Next>*, tal como se muestra en el gráfico 3.2.

A continuación, el utilitario solicitará seleccionar de una lista el tipo de microcontrolador a programar. Aquí se encuentran dos listas desplegables. La primera, de nombre *Family,* permite seleccionar

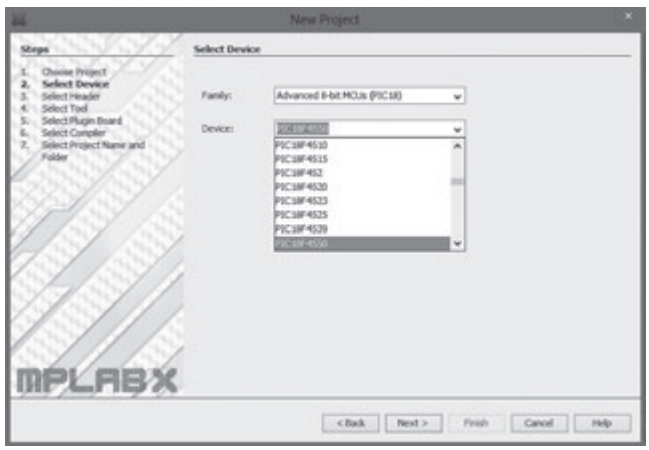
la familia de CPU de Microchip a la cual pertenece el microcontrolador. El PIC18F4550 a utilizar como modelo de este libro pertenece a la familia *Advanced 8-bit MCUs (PIC18)*. Luego, la lista desplegable *Device* tendrá todos los microcontroladores de esta familia. En esta lista se deberá buscar al PIC18F4550, y luego seleccionarlo. Finalmente, se pulsa el botón *Next>*, tal como se muestra en el gráfico 3.3.

Gráfico 3.2. Ventana de inicio para la creación de un nuevo proyecto



Elaboración propia.

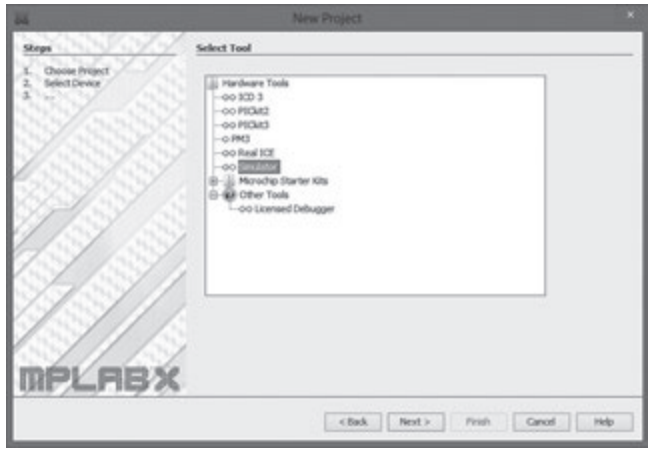
Gráfico 3.3. Selección del modelo de microcontrolador



Elaboración propia.

El siguiente paso consiste en seleccionar el tipo de grabador o simulador a utilizar. En el gráfico 3.4 se aprecia que algunos de los grabadores que se mencionaron en el primer capítulo aparecen en la lista, como ICD3, PICkit2, PICkit3, PM3 y Real ICE. La opción *Simulator* permitirá habilitar los botones de simulación del programa cuando se haya terminado la programación del código. Aquí se selecciona la opción *Simulator* y se pulsa el botón *Next>*.

Gráfico 3.4. Selección de la herramienta de grabación y simulación del proyecto

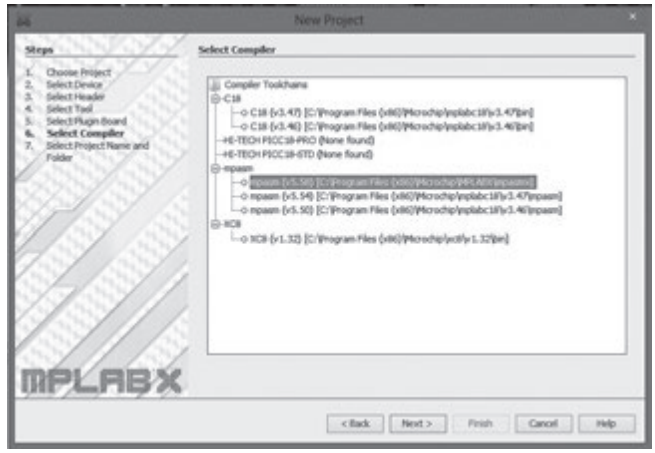


Elaboración propia.

El siguiente paso consiste en seleccionar la herramienta o compilador a utilizar. Para esto, se tiene una serie de opciones que aparecen en la ventana desplegable de nombre *Select Compiler*, tal como se muestra en el gráfico 3.5. No necesariamente todos los compiladores se encontrarán disponibles, ya que requieren ser instalados de forma independiente cada uno (por ejemplo, el compilador XC8). El único compilador por defecto que se encuentra instalado junto con el MPLAB X es el MPASM (*Microchip Assembler* o lenguaje ensamblador de Microchip), con el cual es posible compilar un proyecto programado en lenguaje ensamblador. Esto se observa en el gráfico 3.5. Allí se deberá seleccionar la opción más reciente del compilador (la versión 5.58, para este ejemplo) y nuevamente se deberá hacer clic con el ratón sobre el botón *Next*>.

Posteriormente, se deberá seleccionar la ruta del proyecto donde todos los archivos generados por el ensamblador y el enlazador serán almacenados. MPLAB X creará una carpeta con el nombre del proyecto seguido de las letras .X. Dentro de la carpeta del proyecto se encuentran otras subcarpetas con archivos que describen la configuración del proyecto. Cabe mencionar que la carpeta puede ser movida a otra ruta del disco duro, y el proyecto podrá ser abierto desde esa ubicación por la herramienta sin ningún problema. En el gráfico 3.6 se puede observar la opción de ingreso de la nueva ruta del proyecto y el nombre de este. Se coloca, como nombre de proyecto, «Ejemplo». Finalmente, se deberá pulsar el botón *Finish* para dar por terminada la edición del proyecto. Hasta este momento se habrá creado una carpeta de nombre Ejemplo.X en la ruta especificada. En caso de que la ruta donde se grabe la carpeta del proyecto contenga varias subcarpetas, se deberá verificar que el nombre de cada una no tenga espacios en blanco, ya que MPLAB X tiende a generar errores.

Gráfico 3.5. Selección del compilador para la edición del proyecto



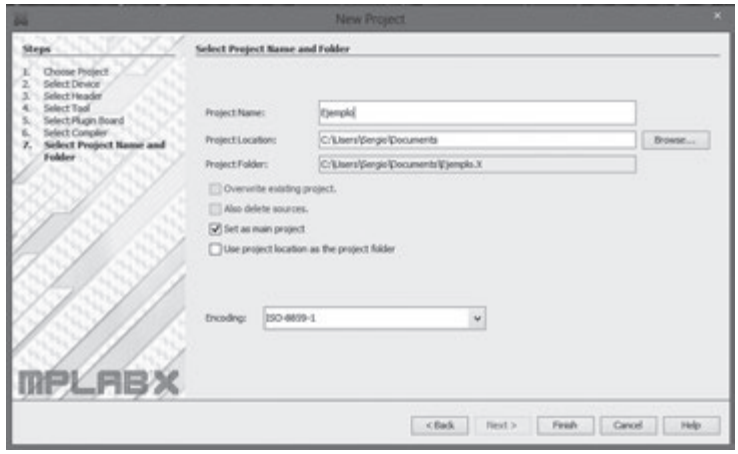
Elaboración propia.

Una vez cumplidos todos los pasos, aparecerá una ventana de resumen del proyecto creado, tal como se muestra en el gráfico 3.7. Aquí se mostrará la ventana principal del MPLAB X con el nuevo proyecto listo para agregarle nuevos archivos y crear la aplicación que el usuario desee. En la ventana desplegable izquierda aparecerá una pestaña de nombre *Projects*, donde se ven una serie de subcarpetas del proyecto creado. Esta ventana se conoce con el nombre de «panel de archivos del proyecto».

3.3 Elaboración de un programa sencillo. Partes del código y reglas básicas

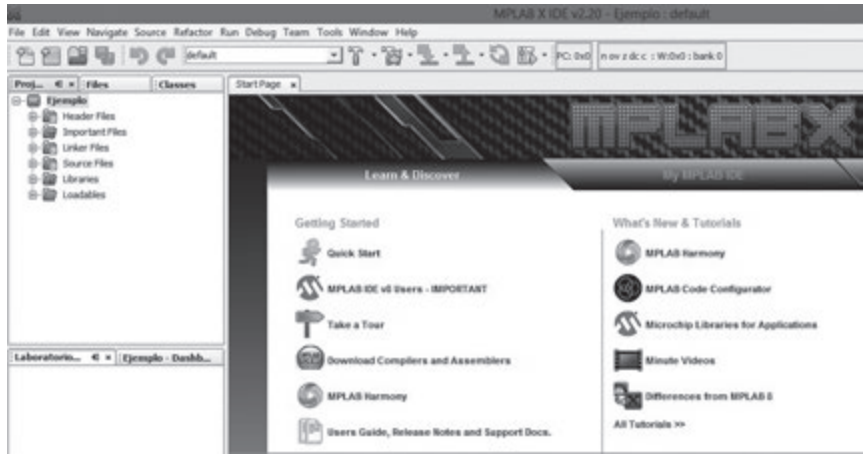
Para elaborar un programa simple, el primer paso consiste en crear una nueva ventana de edición en MPLAB X. Para esto, se ha de ingresar al menú *File->New*, para que aparezca una nueva ventana de selección del formato de archivo a agregar al proyecto. En el gráfico 3.8 se observan dos listas. La primera lista de la izquierda, de nombre *Categories*, contiene carpetas cuyos nombres indican el tipo de archivo a crear. Se selecciona la carpeta *Assembler* pulsando el botón izquierdo del ratón. En la lista desplegable de nombre *File Types* se podrán visualizar tres archivos: *AssemblyFile.asm*, *AssemblyFile.s* y *AssemblyFile.inc*. De esta lista se deberá seleccionar el primer archivo y pulsar sobre el botón *Next>*, tal como se muestra en el gráfico 3.8.

Gráfico 3.6. Selección de la ruta del proyecto y nombre del mismo



Elaboración propia.

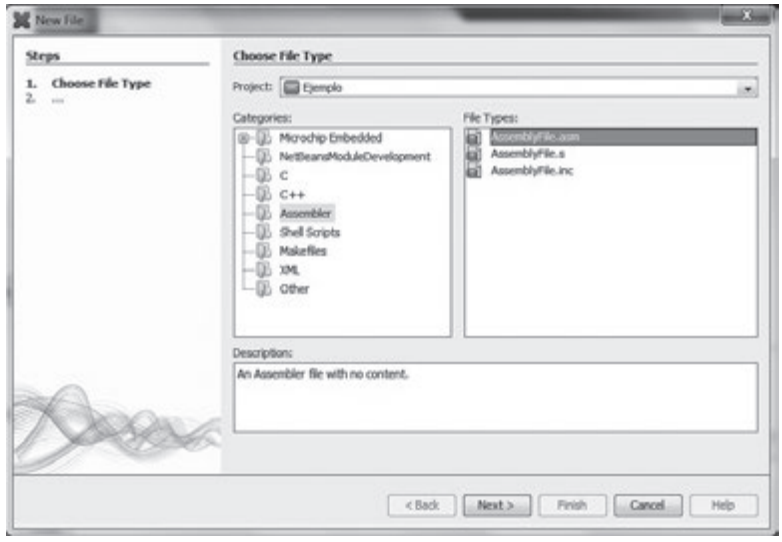
Gráfico 3.7. Ventana inicial del proyecto recientemente creado



Elaboración propia.

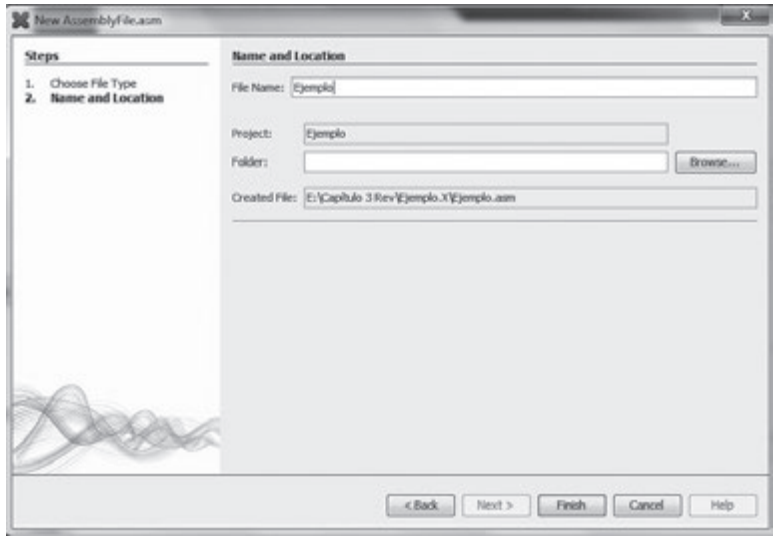
Posteriormente, aparecerá una segunda ventana, como la que se muestra en el gráfico 3.9. Como en el paso anterior se eligió un archivo vacío, en lenguaje ensamblador de extensión .asm, el nombre de archivo a ingresar en la caja de texto, de nombre *File Name* por defecto, llevará la extensión .asm y estará almacenado en la carpeta del proyecto autogenerada. Se coloca de nombre «Ejemplo» y se pulsa el botón izquierdo del ratón sobre *Finish*.

Gráfico 3.8. Ventana de creación de un nuevo archivo para agregar al proyecto



Elaboración propia.

Gráfico 3.9. Definición del nombre de archivo a agregar al proyecto



Elaboración propia.

Luego se deberá ingresar el programa, escribiendo cada línea de forma ordenada. El compilador toma muy en cuenta el orden de las tabulaciones del código (indentar el código), si bien es cierto que no genera errores, es responsabilidad del programador mantener el orden en su código, a fin de que este sea legible. A continuación, en el programa 3.1 se presenta el código que se deberá ingresar en la ventana de texto de nombre Ejemplo.asm.

Programa 3.1. Ejemplo básico de un programa en ensamblador para PIC18F

```
1         #include <P18F4550.INC>        ;Definicion de SFR para el procesador
2;******* Configuración del Oscilador **********
3         CONFIG FOSC   =  HS            ;Oscilador externo de alta frecuencia
4;******** Otros bits de configuracion **********
5         CONFIG PWRT     = ON           ;PWRT habilitado
6         CONFIG BOR      = OFF          ;Brown out reset deshabilitado
7         CONFIG WDT      = OFF          ;Watch Dog Timer deshabilitado
8         CONFIG MCLRE   = ON            ;MCLR como entrada de Reset externo
9         CONFIG PBADEN = OFF            ;Todos los pines como entradas digitales
10        CONFIG LVP      = OFF          ;Programación en bajo voltaje apagado
11;********* Bits de protección ******************
12        CONFIG CP0      = OFF          ;Los bloques del código de programa
13        CONFIG CP1      = OFF          ;no están protegidos
14        CONFIG CP2      = OFF
15        CONFIG CP3      = OFF
16        CONFIG CPB      = OFF          ;Sector Boot no está protegido
17        CONFIG CPD      = OFF          ;La EEPROM no está protegida
18;***************************************************************************
19 CBLOCK 0x000
20                aux
21                var
22 ENDC
23
24 ORG 0x0000
25        goto MAIN
26
27 ORG 0x0020
28MAIN:
29        clrf  TRISB,0
30        movlw 0xFF
31        movwf TRISD
32        clrf  aux,0
33INICIO:
34        movf PORTD,w
35        movwf aux,0
36        incf aux
37        movf aux,W
38        movwf var,0
39        goto  INICIO
40
41        END
```

Antes de explicar el programa 3.1, se debe tener en claro que todo programa se divide en dos tipos de sentencias: instrucciones y directivas. Las primeras corresponden a cualquier sentencia que representa una acción que será ejecutada por parte del procesador del PIC18F. Las directivas, en cambio, son sentencias que representan información al compilador, indicando cómo debe interpretar el código o qué librerías deberá utilizar para ejecutar el programa.

Es interesante notar que el editor de texto del MPLAB X contiene una enumeración en la parte izquierda, permitiendo identificar el número de línea de cada sentencia. Esto es muy necesario cuando el compilador detecta errores, ya que así rápidamente se puede identificar la sentencia que genera el error.

En la línea 1 se encuentra la sentencia *#include <P18F4550.INC>*, la cual es una directiva que permite incluir en el proyecto el archivo P18F4550.INC. Este archivo se encuentra dentro de la siguiente ruta de instalación: C:\Program Files(x86)\Microchip\MPLABX\MPASM Suite. Todos aquellos archivos que se quieran incluir por defecto en esta carpeta deberán estar enmarcados dentro de los operadores <>. En caso de que se desee incluir en el proyecto un archivo en una ruta diferente, se deberá colocar la directiva de la siguiente manera: *#include "C:\PIC18\P18F4550New.INC"*, donde la ruta del archivo se incluye entre las comillas dobles (" "). El archivo P18F4550.INC contiene una serie de declaraciones de registros del microcontrolador PIC18F45550 asociados a cada una de sus respectivas direcciones en el mapa de memoria. Si este archivo no se incluyera en la cabecera, entonces el programador tendría que recordar la dirección y posición de cada bit en cada registro del microcontrolador, lo cual haría demasiado tediosa la programación. El punto y coma (;) indica que lo que se encuentre escrito a la mano derecha es un comentario y, por lo tanto, no va a ser considerado en el proceso de compilación. Es posible escribir todo lo que se desee después del punto y coma, siempre y cuando se mantenga el comentario dentro de la misma línea. El MPLAB X detecta automáticamente los comentarios y los presenta en color verde. Es recomendable el uso de comentarios durante la programación, ya que esto mejora la legibilidad del programa y permite que otros programadores entiendan rápidamente la secuencia del código. Por ejemplo, en la fila 2 se observa un comentario entre «*» indicando que la directiva inferior configurará el reloj del microcontrolador.

Entre las filas 3 y 17 se encuentran las opciones de configuración inicial del microcontrolador. Estas opciones pueden ignorarse, pero el programa adoptaría una configuración por defecto indicada en los bits de configuración del proyecto. Por esta razón, es recomendable utilizar las directivas que a continuación se mencionan, para que luego no haya problemas de compatibilidad con la configuración del *hardware* elegida. Las opciones de configuración elegidas son las siguientes:

1. CONFIG FOSC = HS. Esta opción permite definir el tipo de modo de reloj que el microcontrolador utilizará. HS especifica un modo de alta velocidad para cristales de más de 4 MHz de frecuencia. Las opciones de reloj son las siguientes:

a. HS: Alta velocidad (High Speed) se utiliza para indicar que el cristal externo se encuentra en el rango de los 4 a 20 MHz.
b. HS_PLLHS: es el indicador de que el cristal externo se encuentra en el rango de los 4 a 20 MHz, pero la señal de reloj será conectada al PLL interno del microcontrolador. Esto requerirá el uso de un divisor de frecuencia previo a la entrada del PLL.
c. XT_XT: se indica que se está utilizando un cristal de 4 MHz o menor.

d. XT_PLLXT: se indica que se utiliza un cristal de 4 MHz o menor cuya señal de salida se conecta al PLL interno.
e. INTOSC_HS: se utiliza un oscilador RC interno del microcontrolador como fuente de reloj para el CPU y sus periféricos. Sin embargo, el puerto USB utilizará un reloj entre 4 a 20 MHz.

2. CONFIG PWRT =ON u OFF. El PWRT (de las siglas en inglés *Power Up Timer*) es un temporizador que permite fijar un tiempo de arranque del sistema una vez que se ha aplicado un pulso de Reset. El PWRT posee un contador de 11 bits que utiliza como fuente de reloj el RC interno del PIC18F, generando un tiempo aproximado de 65.5 milisegundos en estado de Reset.
3. CONFIG BOR = ON u OFF. Es la opción que permite habilitar el Reset cuando la tensión de alimentación del PIC es inferior a un nivel de voltaje denominado VBOR. Si se deshabilita esta opción, el PIC18F4550 operará desde 5.5 V hasta 2.7 V.
4. CONFIG WDT = ON u OFF. Esto permite habilitar o deshabilitar el Perro guardián.
5. CONFIG MCLRE = ON u OFF. El pin MCLR se puede configurar para recibir un pulso de Reset externo o como pin RE3 de entrada o salida del Puerto E.
6. CONFIG PBADEN = ON u OFF. Permite definir si todos los pines que operan como entradas analógicas estarán configuradas para este fin o se podrán utilizar como entradas y salidas digitales. Para el último caso, se debe seleccionar la opción OFF.
7. CONFIG LVP = ON u OFF. Especifica la opción de programación en nivel de bajo voltaje.
8. CONFIG CP0 = ON u OFF. Activación o desactivación de la protección del bloque de memoria de programa CP0.
9. CONFIG CP1 = ON u OFF. Activación o desactivación de la protección del bloque de memoria de programa CP1.
10. CONFIG CP2 = ON u OFF. Activación o desactivación de la protección del bloque de memoria de programa CP2.
11. CONFIG CP3 = ON u OFF. Activación o desactivación de la protección del bloque de memoria de programa CP3.
12. CONFIG CPB = ON u OFF. Activación o desactivación de la protección del bloque de sector de arranque del programa.
13. CONFIG CPD = ON u OFF. Activación o desactivación de la protección de la memoria EEPROM.

Existen más opciones de configuración para el microcontrolador. Hasta este punto se han mostrado las básicas. Sin embargo, si se ingresa a la opción *Window>PIC Memory Views->Configuration bits* se obtendrá la lista completa de bits de configuración para todas las opciones disponibles en el sistema embebido que se esté utilizando en el proyecto.

Entre las líneas 19 y 22 se definen los alias a usar en el programa. En este caso se han creado los alias *aux* y *var*, los cuales son equivalentes a las cifras 0x000 y 0x001, respectivamente. Al definir el comando *cblock 0x000*, se está indicando que el primer alias declarado tomará el valor de 0x000; la segunda, el valor de 0x001; la tercera, el de 0x002, y así sucesivamente. La instrucción *endc* define la finalización del bloque de declaración de alias. Es posible declarar la cantidad elevada de alias, sin embargo, esto tiene relación directa con la función que se realice en el programa.

En la línea 23 se ha dejado un espacio en blanco. El programador es libre de dejar la cantidad de espacios en blanco que crea conveniente. El compilador ignora los espacios en blanco, así que esto no afecta el rendimiento del programa, pero sí la visualización y orden del mismo.

En la línea 24 se observa la directiva *ORG 0x000*, la cual define que la siguiente instrucción (ubicada en la línea 25) se encontrará guardada en la dirección 0x000 de la memoria de programa. Esta dirección es especial, ya que se conoce como el Vector de Reset, posición donde el Contador de programa (PC) apunta cada vez que el código comienza a ser ejecutado.

En la línea 25 se cuenta con la primera instrucción ejecutada por el CPU del microcontrolador: *goto MAIN*. La instrucción de salto *goto* indica al Contador de programa que apunte a la dirección dada por la etiqueta MAIN que hace referencia a la dirección 0x020. En otras palabras MAIN, es un alias a la cifra 0x020.

En la línea 27 se encuentra la directiva ORG 0x020. Esta directiva indica que las instrucciones del programa que se encuentren líneas abajo estarán a partir de esta dirección en la memoria de programa. Posteriormente, entre las líneas 29 y 39, se tienen las siguientes instrucciones:

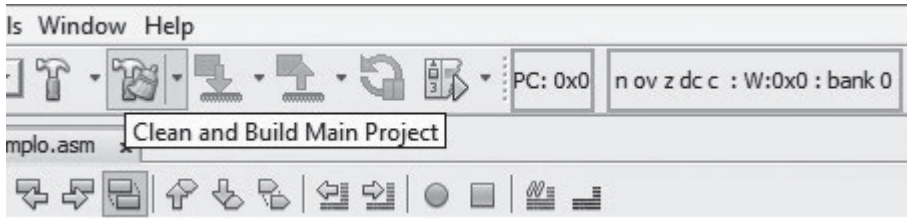
29. *clrf TRISB,0;* Esta instrucción pone los 8 bits del registro TRISB en nivel '0'. El dígito 0 después de la coma indica que la dirección del registro TRISB se considera absoluta. Esta instrucción configura los 8 pines del Puerto B como salida digital.
30. *movlw 0xFF;* Esta instrucción carga el valor literal 11111111_2 en el registro de trabajo W.
31. *movwf TRISD*; Esta instrucción carga el registro de 8 bits TRISD con el valor contenido en W, es decir, 11111111_2. Esto implica que todos los pines del Puerto D han sido configurados como entrada digital.
32. *clrf aux,0*; Esta instrucción pone los 8 bits de la dirección *aux* (en este caso, como *aux* representa el valor 0x00 para el compilador, la dirección de memoria 0x00 será la afectada) en estado '0' lógico. El operando 0 indica que la dirección es absoluta, es decir, se está refiriendo a la dirección 0x00 de todo el mapa de memoria de la SRAM.
33. *INICIO:* Esta etiqueta indica que la dirección 0x028 de la memoria de programa contiene el alias de INICIO.
34. *movf PORTD,W*; El contenido del registro PORTD (configurado como entrada digital) se carga al registro de trabajo W.
35. *movwf aux,0*; El contenido del registro W se carga a la dirección 0x00 (representada por el alias *aux)*, tomando como referencia la dirección absoluta.
36. *incf aux*; El contenido de la dirección especificada por *aux* se incrementa en un dígito LSB.
37. *movf aux,W*; El contenido de la dirección *aux* se carga en el registro de trabajo W.
38. *movwf var,0*; El contenido del registro de trabajo se carga en la dirección *var* (dirección 0x01 absoluta).
39. *goto INICIO*; El Contador de programa toma el valor 0x028 y se repiten las instrucciones 34 a 39.

Finalmente, en la línea 41 se observa la directiva *END*, la cual indica la finalización del programa. Cualquier instrucción escrita debajo de esta directiva será ignorada por el compilador. Es importante notar que esta directiva tiene que estar obligatoriamente al final del código para que el compilador pueda dar un resultado satisfactorio.

El siguiente paso consiste en compilar y enlazar el proyecto. Para esto se deberá pulsar el botón izquierdo del ratón sobre el ícono del martillo y la escoba (*Clean and Build Main Project*), tal como se muestra en el gráfico 3.10. El resultado se observará en la ventana de salida (*Output*) que se encuentra en la parte inferior de la ventana principal del MPLAB X.

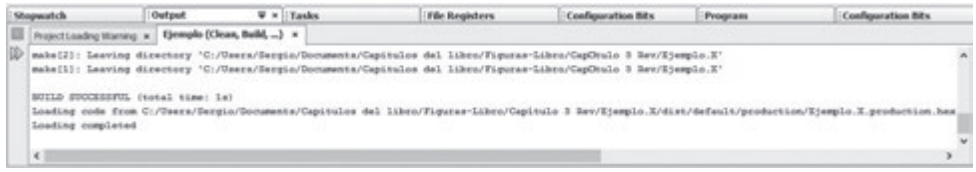
Posteriormente, la ventana de salida mostrará los resultados del proceso. Aquí, como objetivo, se deberá esperar el mensaje «BUILD SUCCESSFUL», tal como se muestra en el gráfico 3.11. Aunque este mensaje implica el éxito en el proceso de compilación y enlace, es importante visualizar en la parte superior la aparición de advertencias (*Warnings*), que en muchas ocasiones tienen información relevante que puede alterar el funcionamiento del programa.

Gráfico 3.10. Botón de compilación *Clean and Build Main Project*



Elaboración propia.

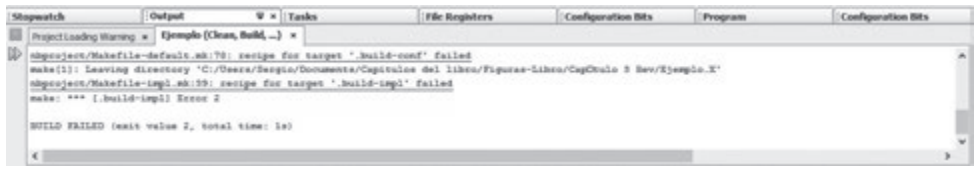
Gráfico 3.11. Mensaje de compilación y enlace satisfactorio



Elaboración propia.

De resultar erróneo el proceso de compilación y enlace, aparecerá un mensaje de error «BUILD FAILED», tal como se muestra en el gráfico 3.12. Esto implica la presencia de uno o varios errores en el ingreso de una instrucción o la invocación a una rutina que no se encuentra existente. Afortunadamente, la ventana de salida que se encuentra en el panel de tareas del MPLAB X indica en qué línea se encuentran los errores, de manera que el programador pueda ubicarlos con facilidad y, de esta forma, corregirlos inmediatamente. En algunos casos, el error no se encuentra en la línea indicada, sino más bien es una consecuencia de un error anterior ubicado en otra posición. En este caso, la experiencia del programador es sumamente importante para ubicar la falla lo antes posible.

Gráfico 3.12. Mensaje de compilación y enlace errado del panel de tareas



Elaboración propia.

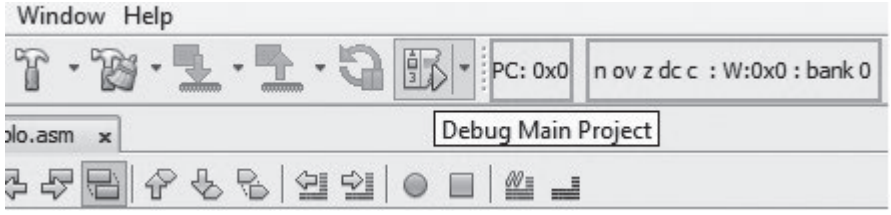
3.4. Simulación de un programa en lenguaje ensamblador

La simulación es un proceso sumamente importante para poder verificar si el programa realizado realmente efectúa las operaciones que el programador ha definido. El MPLAB X cuenta con un simulador que permite realizar los siguientes procesos:

1. Ejecutar cada instrucción paso a paso.
2. Ejecutar las instrucciones hasta llegar a un punto de parada (*Breakpoint*).
3. Observar el resultado de la operación visualizando los contenidos de los registros en una ventana de simulación de registros del PIC18F.
4. Simular el ingreso de valores binarios en los puertos de entrada y salida.
5. Observar el funcionamiento de las interrupciones, activando las banderas de interrupción manualmente.
6. Modificar el valor interno de los registros de configuración de los periféricos y registros de memoria SRAM.

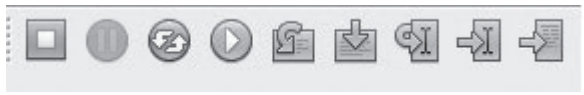
Para iniciar una simulación se tomará como base el programa 3.1, presentado en el subcapítulo anterior. El primer paso para comenzar a simular consiste en ingresar a la opción de depuración, pulsando el botón izquierdo del ratón sobre el botón *Debug Main Project*, tal como se muestra en el gráfico 3.13. Con esto se habilitará una barra de botones con opciones de simulación (gráfico 3.14).

Gráfico 3.13. Invocación de los botones de simulación del programa



Elaboración propia.

Gráfico 3.14. Botones de simulación del programa



Elaboración propia.

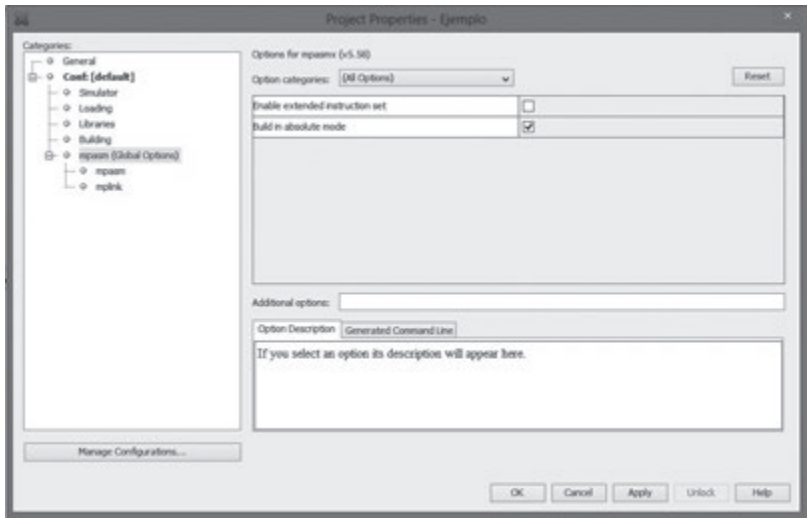
La función de los botones es la siguiente:

1. Botón *Finish Debugger Session*. Este botón permite finalizar el proceso de simulación; de esta manera, la visibilidad de todos los botones desaparecerán de la barra de herramientas.
2. Botón *Stop*. Este botón permite detener la simulación en curso.
3. Botón *Reset*. Esta opción permite inicializar la simulación desde el comienzo del programa, colocando el puntero de programa en la dirección 0x0000.

4. Botón *Run*. Este botón permite iniciar la simulación del programa en ejecución. Aquí existen dos posibilidades: que la simulación sea continua o que simule hasta la detección de un *Breakpoint*, culminando el proceso una vez alcanzado este punto.
5. Botón *Step Over*. Esta opción permite simular la ejecución de una línea de comando cada vez que es pulsado. Pero, al encontrarse ante una instrucción *call*, esta será pasada por alto y se continuará simulando la siguiente instrucción después de la operación *Return*. Muchas veces esta opción es requerida, sobre todo cuando la rutina *call* es repetitiva, como en las operaciones de retardo.
6. Botón *Step Into*. Este botón permite simular la ejecución de una línea de comando cada vez que es pulsado, pero con la diferencia de que, al encontrarse ante una instrucción *call*, se seguirá simulando la ejecución de cada una de las instrucciones dentro de dicha rutina.
7. Botón *Run to Cursor*. Este botón es muy similar al *Run*, con la diferencia de que detendrá la simulación del programa en la ubicación del cursor en el archivo de texto donde se encuentra el código.
8. Botón *Set PC at Cursor*. Este botón permite fijar la posición del contador de programa con la dirección de la memoria FLASH que contiene la instrucción donde se encuentra el cursor en el archivo de texto del código.
9. Botón *Focus Cursor at PC*. Este botón permite alinear el cursor del texto del código en la posición donde se encuentra el puntero de programa (en la simulación).

Para la visualización de los RFE, alias a registros y direcciones de memoria, es necesario configurar las propiedades del proyecto en modo absoluto, para que, durante la simulación, se puedan observar todos los contenidos de los registros usados en el código. Para esto, se deberá pulsar el botón derecho del ratón sobre el nombre del proyecto (Ejemplo, dentro de la lista izquierda en el panel de archivos) y seleccionar la opción *Ejemplo->Set Configuration->Customize* para abrir la ventana de propiedades del proyecto que se muestra en el gráfico 3.15. En esta ventana se deberá seleccionar en la lista *Categories*, la opción *mpasm (Global Options)* y, en la lista *Option categories*, colocar un *check* sobre la opción *Build in absolute mode*. Luego se pulsará el botón izquierdo del ratón sobre el botón *Apply* y luego *OK*, para que la ventana se cierre y la configuración quede guardada.

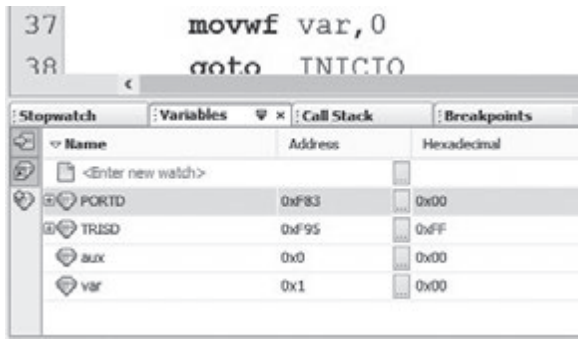
Gráfico 3.15. Configuración del modo absoluto para la visualización de los alias a registros definidos en el programa dentro de la ventana de variables



Elaboración propia.

Luego, se pulsará el botón izquierdo del ratón sobre el ícono *Debug Main Project*, para iniciar la simulación. Dentro del panel de tareas se encuentra una pestaña con el nombre *Variables* (gráfico 3.16). En dicha pestaña, por defecto, aparecerán tres títulos: *Name*, *Address* y *Hexadecimal*. Debajo del título *Name* está el mensaje *<Enter new watch>*. Aquí se deberá escribir el nombre del RFE, alias o dirección absoluta de memoria SRAM que se desea agregar a la lista. Una vez que sea presionado el botón *Enter* del teclado, el registro aparecerá en la parte inferior. Por ejemplo, en el gráfico 3.16 se observa que se han ingresado los registros PORTD, TRISD y los alias *aux* y *var*. El símbolo + al lado izquierdo de los RFE implica que se pueden expandir para visualizar sus bits de manera individual. Debajo del título *Address* se observará la dirección que ocupa cada registro en el mapa de memoria de la SRAM, y debajo del título *Hexadecimal* se podrá hallar el valor contenido por cada registro en base hexadecimal. Son estos últimos valores los que se actualizarán con la simulación del programa. El gráfico 3.16 muestra los valores por defecto al momento del inicio del programa.

Gráfico 3.16. Ventana variables para la visualización de RFE, registros alias y direcciones de memoria absolutas



Elaboración propia.

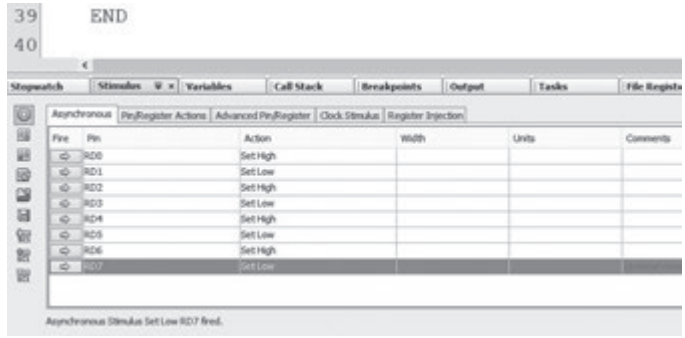
Para poder simular el programa 3.1, es necesario que el Puerto D tome un valor inicial para simular el efecto del cálculo que se realiza sobre su lectura. Para esto, se deberá ingresar a la opción *Window->Simulator->Stimulus*, lo cual abrirá la lista *Stimulus* que se observa en el gráfico 3.17. Dentro de la pestaña *Asynchronous* se ha de agregar cada pin que se desea afectar. En este caso son los ocho pines que corresponden al registro PORTD: RD0, RD1, RD2, RD3, RD4, RD5, RD6 y RD7. Al lado derecho de cada pin se encuentra una lista desplegable debajo el título *Action*, donde se podrá elegir si dicho pin se colocará en los siguientes estados:

1. *Set High*. Permite fijar el pin al estado '1'.
2. *Set Low*. Permite fijar el pin al estado '0'.
3. *Toggle*. Niega el valor que el pin tenía anteriormente.
4. *Pulse High*. Permite fijar un estado de '1', pero por un tiempo determinado. La magnitud del pulso se fija en la columna "Width" y las unidades en la columna "Units". Las unidades pueden ser ciclos, nanosegundos, microsegundos, milisegundos y segundos.
5. *Pulse Low*. De la misma manera que en el caso anterior, se puede fijar un pulso de estado '0' fijando la magnitud y las unidades de tiempo.

Una vez definido el estado de los ocho pines, se deberá pulsar el ícono con el dibujo de la flecha apuntando a la derecha de cada pin, lo cual disparará dicho estado en la simulación del registro PORTD. El resultado del disparo se visualizará en la parte inferior de la ventana.

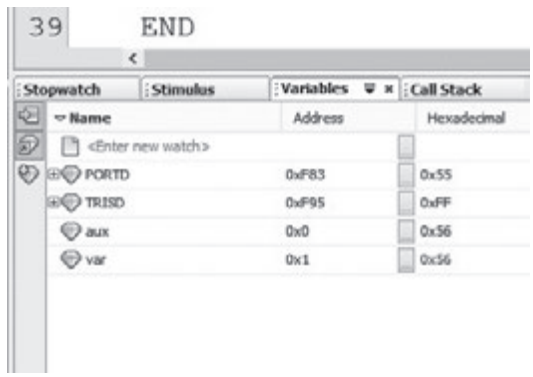
Es muy importante que se haya definido previamente a la simulación un *Breakpoint* en alguna instrucción del programa, para que, cuando se inicie la simulación, esta se detenga en un punto conocido. Una vez hecho esto, se podrá pulsar el botón *Step Into* para simular cada línea de instrucción y observar el resultado inmediato en la pestaña *Variables*. Por ejemplo, en el gráfico 3.18 se observan los contenidos de los registros alias *aux* y *var* afectados por el valor simulado 0x55 en el registro PORTD.

Gráfico 3.17. Valores finales de los registros y variables obtenidos de la simulación



Elaboración propia.

Gráfico 3.18. Pestaña de *Variables* con los resultados de la simulación afectados



Elaboración propia.

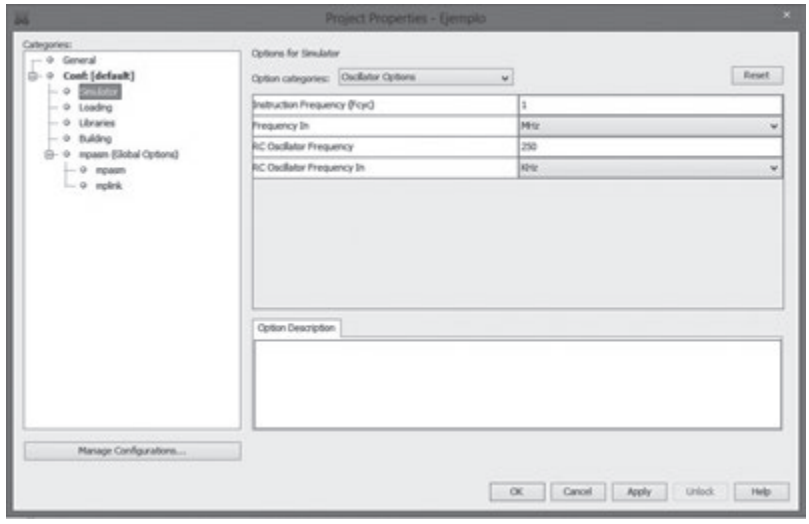
Como se puede observar, al ingresar el valor 0x55 al Puerto D, el registro *aux* tomará dicho valor incrementado en 1, es decir, 0x56. Estos valores finalmente son cargados al registro de trabajo W y al alias *var*. Es posible seguir asignando nuevos valores al Puerto D durante la simulación. La única acción que deberá realizarse para este proceso es cambiar el estado del pin en la columna *Action* y pulsar el botón de disparo para el respectivo pin.

Los estímulos presentados se conocen como «asíncronos», dado que se pueden generar en cualquier momento del programa. Existen, por supuesto, otras opciones para realizar estímulos sincronizados con ciertas etapas del programa o con ciertos valores de registros. Se recomienda al lector probar las otras opciones para que vaya descubriendo otras formas de estímulo. Sin embargo, esto

debería realizarse cuando se culmine el capítulo sobre el lenguaje ensamblador, para tener un mejor entendimiento.

Durante la simulación paso a paso del programa, también es posible llevar un control del tiempo que tarda cada instrucción en ejecutarse. Esto permite saber exactamente los lapsos que cada rutina tarda y, obviamente, ayudará al programador a determinar si la rutina creada es pertinente o no para su aplicación. Para esto, es necesario fijar en la configuración del proyecto pulsando el botón derecho del ratón sobre el nombre del proyecto en el panel de archivos, seleccionando la opción *Set Configuration->Customize->Simulator*. Luego, en la pestaña *Instruction Frequency (Fcyc)*, se deberá ingresar la frecuencia del reloj de instrucciones y debajo, en *Frequency In*, las unidades en MHz.

Gráfico 3.19. Configuración del reloj de simulación del proyecto



Elaboración propia.

Para conocer el tiempo y el número de ciclos de ejecución que tarda el programa, se deberá seleccionar la ventana *Stopwatch*. Para abrir dicha ventana se ingresará a la opción *Window->Debugging->Stopwatch*. La pestaña *Stopwatch* aparecerá en el panel de tareas del proyecto y se verá tal como se muestra en gráfico 3.20.

En la pestaña *Stopwatch* se puede ver la información de los ciclos ejecutados y el tiempo transcurrido. Conforme se pulsa el botón *Step Into*, la pestaña *Stopwatch* irá acumulando el tiempo ejecutado por cada línea de instrucción, según el valor del reloj de instrucciones configurado.

Las casillas de la segunda fila muestran el tiempo transcurrido en unidades de microsegundos. Los botones *Clear History* (ícono en forma de *Papelera de reciclaje*) y *Clear Stopwatch* (ícono en forma de reloj) permiten limpiar la subventana e inicializar el ciclo de conteo en instrucciones en 0 µsegundos. El hecho de haber seleccionado un reloj de instrucciones implica que la frecuencia de reloj del procesador seleccionada será de 4 MHz. Como se sabe, el ciclo de ejecución básico requiere de cuatro ciclos de reloj. Por lo tanto, un ciclo de ejecución utiliza un reloj base de 4÷4 = 1 MHz. Esto implica que cada ciclo tarda $\frac{1}{1MHz} =$ 1 µsegundo. Por ejemplo, la instrucción *incf aux* se ejecuta en un ciclo de instrucción, por lo tanto, tarda 1 µsegundo en ejecutarse. Esto se observa en la ventana *Stopwatch* (gráfico 3.20).

Gráfico 3.20. Pestaña *Stopwatch*

Elaboración propia.

3.5 Los archivos de configuración

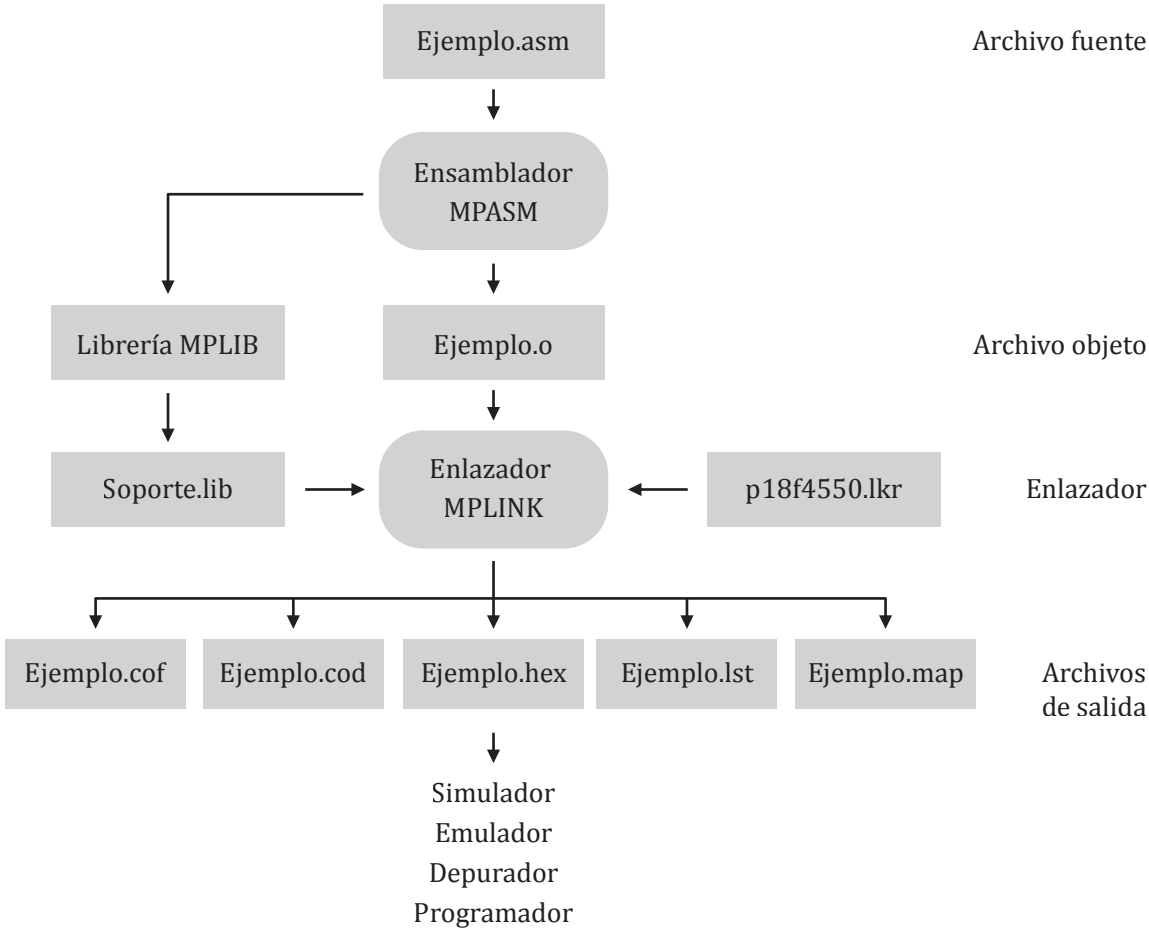
Un proyecto en MPLAB X está formado por una carpeta y un conjunto de subcarpetas y archivos, los cuales se pueden movilizar de una ruta específica a otra sin ningún inconveniente con el IDE al momento de abrir el proyecto. Las subcarpetas involucradas son las siguientes:

1. *MyProject*. Es la carpeta principal del proyecto y contiene al archivo *makefile* y el archivo de texto con el programa. El archivo *makefile* es autogenerado por el proyecto y solo puede ser modificado automáticamente por el IDE.
2. *build*. Esta carpeta contiene los archivos objeto (extensión .o) y los dependientes.
3. *dist*. Contiene los archivos de salida del proyecto, como el de extensión .hex (ejecutable), .coff y .lib (archivos de librerías).
4. *Nbproject.* Contiene los archivos de metadatos del proyecto de extensión .xml.
5. *default*. Contiene las configuraciones del proyecto.
6. *production*. Contiene las carpetas de producción y depuración con los archivos ejecutables del proyecto, que pueden ser descargados en la memoria FLASH del PIC18F o simulados por alguna herramienta de *software*.

En el diagrama presentado en el gráfico 3.21 se muestra el proceso de elaboración de un proyecto, que se inicia con el archivo Ejemplo.asm, el cual contiene el código fuente del programa. Se puede tomar como referencia el programa usado en el programa 3.1, el cual es analizado por el compilador de lenguaje ensamblador MPASM. Si no se encuentran errores de sintaxis, el MPASM producirá un archivo objeto como salida, de nombre Ejemplo.o. Este archivo será tomado como entrada para el enlazador MPLINK, que realizará el proceso de enlace entre el programa fuente y los demás archivos, entre los que se encuentran el archivo específico del modelo de microcontrolador (Dispositivo.lkr) y un archivo de librería (libreria.lib) que es opcional y contiene a varios archivos objeto. Las librerías incluyen rutinas que pueden ser invocadas desde el archivo fuente. Cabe resaltar que, si se genera un error al momento de invocar una

rutina (por ejemplo, si se escribe mal el nombre de la etiqueta que referencia a dicha rutina en el archivo .lib), el problema no será detectado por el MPASM, sino más bien por el MPLINK.

Gráfico 3.21. Elementos del proyecto en MPLAB X con el compilador MPASM



Elaboración propia.

Como resultado del proceso de enlace se obtiene un conjunto de archivos. Estos archivos son los siguientes:

1. **Ejemplo.X.cof**. Este archivo contiene información sobre símbolos y el código objeto del programa. El archivo no es legible si es que se abre con un editor de MPLAB X; más bien, contiene código interno, que debe ser interpretado por el enlazador.
2. **Ejemplo.X.cod**. Este archivo genera información de salida sobre símbolos y opciones de depuración.
3. **Ejemplo.X.err**. Este archivo contiene una lista de errores y advertencias generadas en los procesos de compilación y enlace. Idealmente, debería estar vacío si es que se ha generado el archivo de salida de extensión .hex de forma satisfactoria.
4. **Ejemplo.X.hex**. Este es el archivo que contiene el código máquina, que puede ser grabado directamente en la memoria de programa del microcontrolador. Adicionalmente, este archivo puede ser aprovechado por herramientas de simulación, emulación y depuración.

5. **Ejemplo.X.lst**. Es un archivo de listado generado por el ensamblador MPASM. Detalla la cantidad de líneas, direcciones de la SRAM y símbolos que están siendo usados por el programa.
6. **Ejemplo.X.map**. Es un archivo que contiene la lista de símbolos y etiquetas. Además, especifica los espacios de la memoria de programa que están siendo utilizados por el código.

Los archivos de salida contienen información del programa. El archivo .hex es el que puede ser usado para diversos fines. Como se observa en el gráfico 3.21 estos fines son la simulación, la emulación, la depuración y la programación. Si bien es cierto que estos procesos se parecen mucho, cada uno es diferente. A continuación se define cada uno de ellos, para tener los conceptos en claro:

1. **Simulación.** Es el proceso en el cual se toma como entrada el archivo .hex y se realiza una simulación en *software* del programa. Como ejemplo, se tiene el Simulador del MPLAB X visto en el subcapítulo 3.4, o también el *software* Proteus ISIS, que realiza la simulación del funcionamiento del microcontrolador en *software*.
2. **Emulación.** Es un proceso de simulación también, pero es ejecutado en una herramienta de *hardware* en tiempo real que no necesariamente contiene el microcontrolador objetivo; más bien, el *hardware* puede simular el funcionamiento de dicho sistema embebido. Estas herramientas por lo general son costosas. Microchip cuenta con el MPLAB REAL ICE, que es un emulador para dsPIC y microcontroladores de diversas familias.
3. **Depuración.** La depuración es un proceso de simulación paso a paso que se controla desde el IDE MPLAB X, pero es ejecutado en el microcontrolador objetivo. Para esto se requiere un depurador. El Pickit 2 y el Pickit 3 son programadores y depuradores, ya que permiten realizar este proceso. El MPLAB X puede, entonces, simular el estado de las variables internas, leyendo su contenido dentro del sistema embebido y mostrando sus valores en las pestañas *Variables* o *File Registers*.
4. **Programación.** Este proceso consiste en cargar el archivo .hex en la memoria de programa del microcontrolador. Un programador realiza este proceso y permite que el sistema embebido posteriormente pueda ejecutar el programa de manera autónoma (*Stand alone*).

3.6 El compilador MPLAB XC8

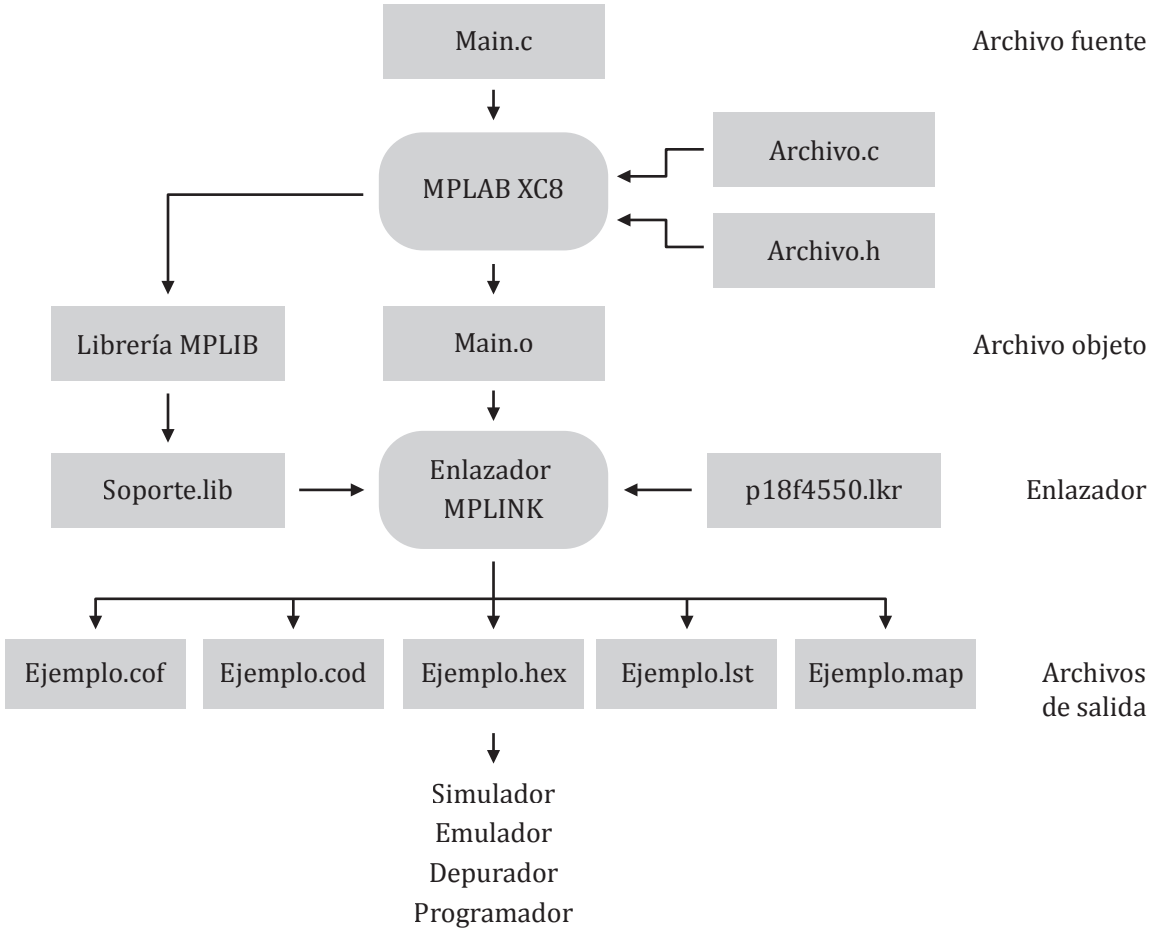
El compilador MPLAB XC8 es una herramienta que se puede descargar de manera gratuita de la página web de Microchip . El archivo de descarga es un ejecutable que, al ser instalado en el computador (previamente, el MPLAB X IDE debe estar instalado), añade la opción de compilación del lenguaje ANSI C en el proyecto. Esta herramienta garantiza el proceso de optimización de código (modo PRO) durante los primeros 60 días de instalado. Posteriormente, se pierde la optimización, pero se puede seguir utilizando el compilador para crear nuevos proyectos en lenguaje ANSI C (modo FREE).

Los archivos de proyecto con el MPLAB XC8 son similares al caso explicado en el subcapítulo anterior. Inclusive, el proceso de enlace es el mismo (se utiliza la herramienta MPLINK para tal fin), tal como se observa en el diagrama del gráfico 3.22. En este caso, el compilador toma como entrada un archivo fuente de extensión .c. Este archivo debe incluir una función de nombre *main()*, la cual contiene la secuencia de instrucciones por ser ejecutada. Adicionalmente, pueden agregarse otros archivos con funciones denominadas «librerías», que se componen de un archivo de extensión .c y otro .h. Pueden

existir múltiples pares de estos en el proyecto. Todos estos archivos son tomados por el compilador y convertidos a un archivo objeto de extensión .o.

El proceso de optimización de código consiste en que la conversión de los archivos fuentes al archivo objeto se realice de manera que se minimicen los recursos de memoria de datos y de programa. Uno de los grandes inconvenientes de los compiladores para ANSI C es que muchos programadores se quejan de que podrían desarrollar un código en ensamblador equivalente a uno realizado en ANSI C, que es más eficiente que el generado por el compilador. En muchas ocasiones esto es cierto, sobre todo si el compilador es gratuito, pero el ahorro en tiempo de elaboración de código en ANSI C es muy grande comparado con la eficiencia en el uso de recursos del microcontrolador. Esto se manifiesta cuando el programa crece en complejidad.

Gráfico 3.22. Elementos de un proyecto en MPLAB X con el compilador XC8



Elaboración propia.

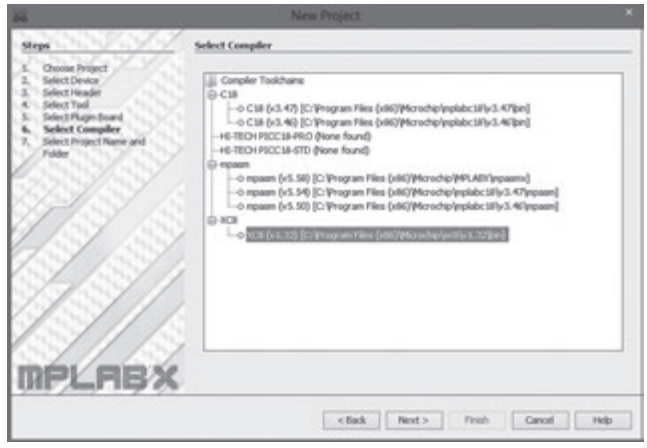
Más adelante en el libro se verán las ventajas del uso del lenguaje ANSI C para la programación de sistemas embebidos y se dedicará un capítulo completo a este lenguaje. Posteriormente, se presentarán aplicaciones elaboradas en lenguaje C. Sin embargo, es imprescindible que el lector se tome un tiempo en instalar el compilador XC8 en su computador, para que pueda hacer uso de los ejemplos dados en este texto.

3.7 Creación de un nuevo proyecto en MPLAB usando el compilador MPLAB XC8

Para crear un nuevo proyecto con el compilador MPLAB XC8 se siguen los mismos pasos vistos en el subcapítulo 3.2, según los procedimientos mostrados entre los gráficos 3.1 y 3.5. Se debe crear un nuevo proyecto con de nombre «EjemploC», para luego repetir los pasos vistos para el ejemplo anterior, pero el paso del gráfico 3.5 deberá ser reemplazado por el del gráfico 3.23. Aquí se puede observar que la herramienta activa que se debe seleccionar es el XC8 versión 1.32 (esta versión es la vigente hasta la edición de este libro). Se pulsa el botón *Next* y se continúa con los siguientes pasos.

Una vez que se ha finalizado la edición del proyecto, se deberá crear un nuevo archivo de programa (mediante la opción *File->New*) y seleccionar un archivo de tipo *C Source File*, como se ve en el gráfico 3.24. Deberá crearse el archivo con el nombre EjemploC también (tomar en cuenta que no es una obligación que el nombre del archivo de programa sea similar al del proyecto). Al colocar el nombre del archivo, no es necesario escribir la extensión .c, ya que esto es realizado por la herramienta de manera automática. Una vez creado el archivo, aparecerá una ventana de texto dentro de la pestaña EjemploC en el panel de edición. Aquí se podrá observar nuevamente que cada línea presenta una numeración para facilitar el seguimiento del código.

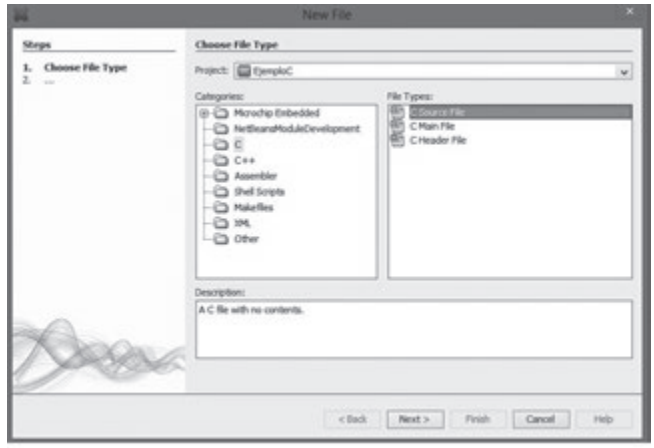
Gráfico 3.23. Ventana de selección del compilador XC8



Elaboración propia.

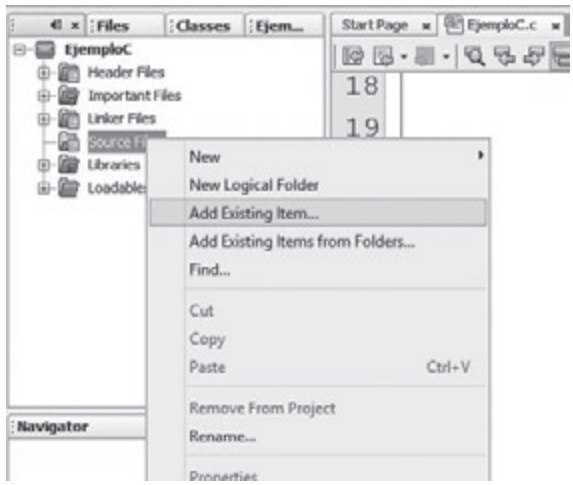
Una vez creado el archivo, se debe agregar al proyecto pulsando el botón derecho del ratón sobre la subcarpeta *Source Files* del panel de archivos y seleccionando la opción *Add Existing Item*..., como se muestra en el gráfico 3.25.

Gráfico 3.24. Ventana de selección de un archivo nuevo para lenguaje ANSI C



Elaboración propia.

Gráfico 3.25. Agregar un archivo de extensión .c al proyecto



Elaboración propia.

A continuación, se debe ingresar un programa equivalente al visto en el subcapítulo 3.3. Este programa realiza las mismas funciones que el programa 3.1, pero se encuentra programado en lenguaje ANSI C.

Programa 3.2. Ejemplo básico en lenguaje ANSI C para el PIC18F4550

```
1#include <xc.h>
2
3#pragma config FOSC        = HS
4#pragma config PWRT        = ON
5#pragma config WDT         = OFF
6#pragma config LVP         = OFF
7#pragma config BOR         = OFF
8#pragma config DEBUG       = ON
9#pragma config VREGEN      = OFF
10#pragma config PBADEN     = OFF
11#pragma config MCLRE      = ON
12
13unsigned char aux, var;
14
15 void main(void)
16 {
17      TRISB = 0x00;       //Puerto B salida digital
18      TRISD = 0xFF;       //Puerto D entrada digital
19      aux   = 0x00;
20      while(1)
21      {
22            aux = PORTD;
23            aux++;
24            var = aux;
25      }
26}
```

El programa es muy sencillo y muestra las partes básicas de un algoritmo creado en lenguaje ANSI C. En la primera línea se incluye el archivo xc.h almacenado en la carpeta *include* dentro de la ruta de instalación del compilador MPLAB XC8. Este archivo incluye la referencia a otros archivos de librería que contienen la definición de los nombres de los registros de los periféricos y bits de estos asociados a las direcciones de memoria correspondientes para el PIC18F4550 y otros modelos más. Este archivo cumple un rol muy similar al archivo P18F4550.INC usado en el ejemplo para lenguaje ensamblador. Además, contiene referencias a funciones estándar.

Al igual que en el lenguaje ensamblador, los espacios en blanco no tienen ningún significado para el compilador. El programador puede dejar tantos espacios en blanco como considere necesarios.

Entre las líneas 3 y 11 se tienen las directivas para la configuración inicial del microcontrolador. Estas opciones fueron explicadas anteriormente y en el programa en ANSI C se mantienen con la misma configuración. Si se observa de manera intuitiva, la única directiva nueva agregada al programa es #pragma DEBUG ON, la cual habilita la opción de depuración del microcontrolador. Existen otras directivas más que son ignoradas. Estas, al no ser usadas, tendrán seleccionadas las opciones por defecto.

Posteriormente, en la línea 13 se declaran dos variables de 8 bits cada una: *aux* y *var*. Estas variables ocupan un espacio en la memoria SRAM del microcontrolador. Las direcciones asignadas se encuentran dadas por el compilador. Por defecto, la dirección de inicio para declaración de variables es la 0x01, pero esto se puede cambiar realizando ajustes de configuración en el proyecto.

En la línea 15 se encuentra la función principal *main()*. Esta función contiene en su interior (delimitado por las llaves { }) todas las instrucciones que serán ejecutadas por el microcontrolador. En un programa en ANSI C, se debe evitar que la ejecución del programa llegue al final de la función *main()*, ya que esto generaría una condición de Reset del microcontrolador y todo el programa se volvería a ejecutar desde un inicio. Es sumamente importante notar que todo programa en ANSI C debe contener una única función de nombre *main()*.

La línea 17 inicializa el registro TRISB con el valor de 0x00, con lo cual todos los pines del puerto B se configuran como salidas digitales. Mientras que en la línea 18, el registro TRISD se carga con el valor 0xFF y sus pines se configuran como entradas digitales. Luego, en la línea 19, la variable *aux* se inicializa en 0x00. El símbolo // tiene el mismo efecto que el punto y coma en lenguaje ensamblador. Todo el texto escrito al lado derecho de // se considera un comentario.

En la línea 20 se establece un bucle *while(1)* delimitado hasta la línea 25 por las llaves ({ }). Este bucle establece una repetitividad de las instrucciones que se encuentran entre las líneas 22 y 24, impidiendo así que el programa llegue al final de la función *main().* La instrucción de la línea 22 toma el contenido del Puerto D y lo guarda en la variable *aux*. La instrucción de la línea 23 incrementa la variable *aux* en 1 dígito y, finalmente, la instrucción de la línea 24 carga el contenido de la variable *aux* a la variable *var*.

El proceso de compilación y enlace se obtiene pulsando el botón izquierdo del ratón sobre el ícono *Clean and Build Main Project* en la barra de herramientas. Una vez que el mensaje de compilación y enlace ha sido satisfactorio (*Build Successful*), se tendrán los archivos de salida EjemploC.X.cof, EjemploC.X.cod, EjemploC.X.hex, EjemploC.X.lst, EjemploC.X.map y EjemploC.X.err.

3.8 Proceso de grabación del microcontrolador PIC18F4550

Para el proceso de grabación se trabajará con un grabador universal de Microchip muy conocido y que se encuentra disponible en el mercado: el PICkit 2. El grabador se muestra en el gráfico 3.26. Además, es posible encontrar en internet diseños alternativos que simulan el funcionamiento de este grabador. Estos diseños se basan en el microcontrolador PIC18F2550 que contienen también un módulo USB interno a través del cual se realiza la comunicación con el *software* MPLAB X.

Gráfico 3.26. Grabador/Depurador PICkit 2

Fuente: http://www.voti.nl/pickit2faq/

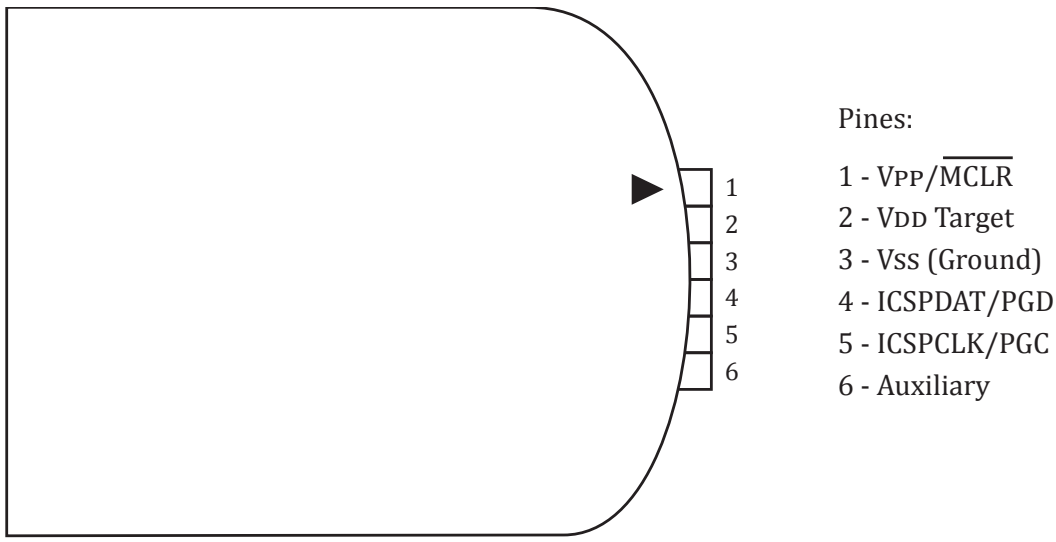
El PICkit 2, en el extremo izquierdo, presenta un conector hembra de 6 contactos, los cuales deben ingresar a un conector macho soldado sobre el circuito impreso que aloja al microcontrolador. La programación del microcontrolador requiere de tres pines: MCLR (o Reset), PGD (Datos) y PGC (Reloj). Adicionalmente, el PICkit 2 es capaz de proveer una tensión de alimentación de 5 V o de 3.3 V para realizar la grabación del PIC18F sin necesidad de una fuente externa. En el extremo derecho, el PICkit 2 cuenta con un conector mini-USB que permite la conexión al computador. Es a través del puerto USB que el grabador recibe la alimentación y es capaz de proveer 5 V de energía al microcontrolador. La capacidad de corriente estará limitada por el puerto USB usado.

El diagrama de pines del PICkit 2 se muestra en el gráfico 3.27. Aquí se puede observar que la flecha (que se observa en la parte superior derecha del gráfico 3.27) marca la posición del pin Reset. El pin VDD puede generar una tensión de 5 V o 3.3 V dependiendo del modelo de microcontrolador. El pin auxiliar no tiene un uso durante la programación o depuración del sistema embebido.

Una vez que se coloca el PICkit 2 en el puerto USB de la computadora, estando instalado previamente el MPLAB X IDE, este grabador será reconocido inmediatamente por el sistema operativo. De esta manera, será posible descargar el archivo .hex directamente a la memoria del programa del sistema embebido simplemente usando el MPLAB X IDE y el PICkit2 conectado al puerto USB.

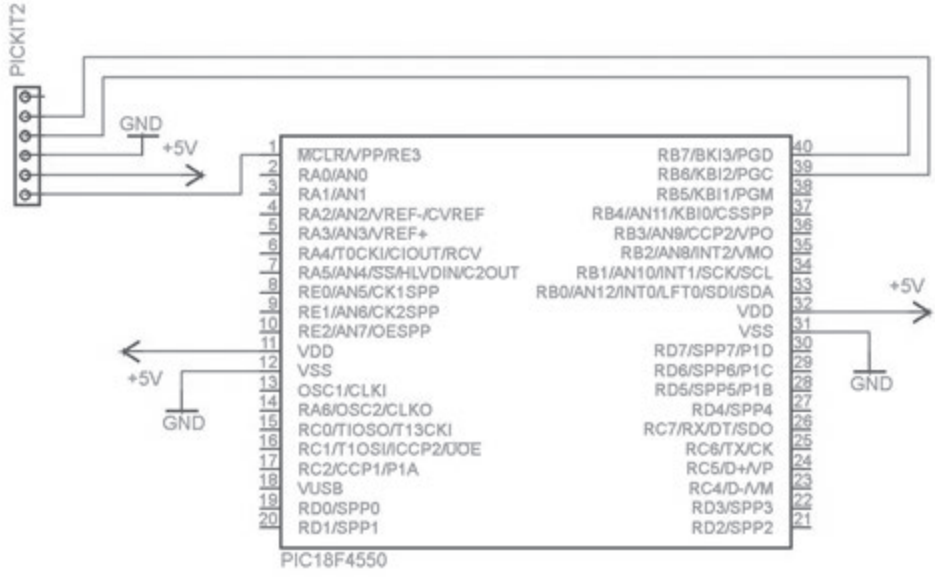
En el gráfico 3.28 se muestra el diagrama de pines del microcontrolador PIC18F4550. Allí se observan también las conexiones mínimas requeridas del microcontrolador para que pueda ser programado o depurado por el PICkit 2. Este proceso también se puede llevar a cabo en una placa de pruebas (*Protoboard)*, ya que no es imprescindible un circuito impreso para esto. Los microcontroladores de Microchip cuentan con un pin Reset denominado MCLR y dos pines de E/S que cumplen con la función PGD y PGC. Estos pines se deben conectar de manera directa a los pines MCLR, PGD y PGC del PICkit 2. De la línea VDD es posible extraer la tensión de alimentación para el sistema embebido. Se puede, también, modificar la tensión generada por el PICkit 2 a través del MPLAB X entre rangos de 1.8 V y 5 V. Por defecto, la tensión es de 5 V. Si hubiera una inadecuada conexión entre PICkit 2 con el microcontrolador, no se dañaría al microcontrolador, pero tampoco se permitiría su grabación.

Gráfico 3.27. Diagrama de pines del PICkit 2



Fuente: Pickit 2 User Guide (http://ww1.microchip.com/downloads/en/devicedoc/51553e.pdf).

Gráfico 3.28. Conexiones del PIC18F4550 al grabador/depurador PICkit 2



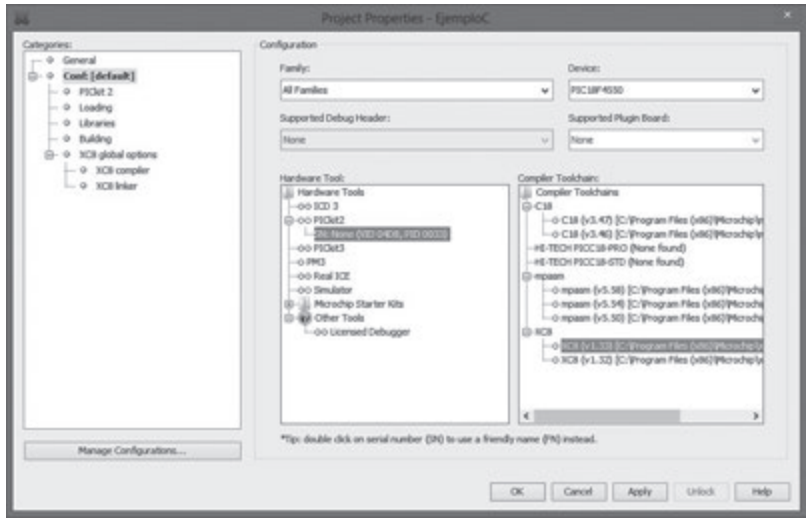
Elaboración propia.

Una vez que se ha conectado el PICkit 2 al microcontrolador, es posible realizar el proceso de grabación. Para esto, primero se deberá modificar la configuración del proyecto, ya que desde un inicio se programó para ser simulado. Para tal fin, se deberá posicionar el cursor del ratón sobre el panel de archivos, pulsar el botón derecho y seleccionar la opción *Set Configuration->Customize* para abrir la ventana de propiedades del proyecto. En dicha ventana se deberá seleccionar la opción Pickit 2 en la lista correspondiente a *Hardware Tools*, como se observa en el gráfico 3.29. Luego, se pulsará con el botón izquierdo del ratón sobre los botones *Apply* y *OK*.

Para poder programar la memoria FLASH del PIC18F4550, es necesario que el programa haya compilado satisfactoriamente. Luego, se pulsará con el botón izquierdo del ratón sobre el ícono *Make and Program Device Main Project* (ícono con una flecha verde apuntando hacia abajo sobre un circuito integrado) de la barra de herramientas, como se muestra en el gráfico 3.30. Esta pulsación generará automáticamente los procesos de compilación, enlace y, luego, la grabación. Una vez finalizado, el programa comenzará a ser ejecutado en el microcontrolador. El ícono a la derecha (con una flecha verde apuntando hacia arriba sobre un circuito integrado) permite realizar el proceso inverso, es decir, leer las memorias del PIC18F4550 (FLASH, SRAM, EEPROM) para observar su contenido en las ventanas del panel de tareas del MPLAB X.

Cada proceso se encuentra acompañado de su respectivo resultado en la ventana de salida. Si cada proceso se ejecuta, la ventana de salida mostrará un mensaje satisfactorio. Por ejemplo, en el gráfico 3.31 se observa el mensaje resultante del proceso de grabación exitoso de la memoria de programa. De ocurrir algún error entre la conexión del PICkit 2 y el microcontrolador, aparecerá un mensaje indicando que el proceso de grabado, borrado o verificación no se pudo dar. Ante esta situación, es recomendable revisar las conexiones del microcontrolador al conector del PICkit 2. Si el proceso de grabación utilizara un grabador/depurador PICkit 3, los pasos de grabación serían idénticos a los ya señalados.

Gráfico 3.29. Configuración del proyecto para la grabación mediante el PICkit 2



Elaboración propia.

Gráfico 3.30. Programación de la memoria FLASH del PIC18F4550

Elaboración propia.

Gráfico 3.31. Mensaje de programación satisfactoria de la memoria de programa

Elaboración propia.

Video

Cómo crear un proyecto en MPLAB-X en lenguaje ensamblador y ANSIC y luego programarlo en el PIC18F4550.

http://bit.ly/1OCSlj1

Resolución de videos en HD

Preguntas de repaso

1. ¿Qué entiende por *Breakpoint*? ¿Para qué se utiliza en la simulación?
2. Enumere los pasos para crear un nuevo proyecto en MPLAB X IDE haciendo uso del compilador MPASM.
3. Identifique las posibles opciones de configuración de reloj para el microcontrolador PIC18F4550.
4. ¿Cuál es la diferencia de un registro alias con un RFE en un programa en lenguaje ensamblador?
5. Explique cuál es la función de la directiva #include <P18F4550.INC> en un programa en lenguaje ensamblador.
6. ¿Qué función cumple la directiva *END* en el lenguaje ensamblador?
7. ¿En qué se diferencia un depurador de un programador?
8. Si el cristal externo conectado al microcontrolador PIC18F4550 es de 12 MHz, ¿cuánto tiempo tarda la instrucción *movlw 0x10* en ejecutarse?
9. ¿El PICkit 2 es grabador, depurador o ambos?
10. ¿Cuál es la utilidad de la función *main()* en un programa en ANSI C?

Respuestas

Respuestas del capítulo 3:

http://bit.ly/1Fw7uNL

Capítulo 4. El lenguaje ensamblador del PIC18F

El lenguaje de programación en ensamblador es un método para escribir programas haciendo uso de instrucciones que tienen un equivalente directo en código máquina. La sintaxis de cada instrucción está estructurada de tal manera que permite la conversión directa al código máquina y se hace entendible por el programador, ya que el mnemónico de cada instrucción posee un nombre que por lo general contiene las siglas de la operación que realiza. Por ejemplo, la instrucción *addlw 0x30* fácilmente puede ser interpretada por el programador, ya que da a entender que hay una suma de por medio (*add*, que en inglés significa «sumar») que involucra al registro W y un factor hexadecimal de 0x30.

Como todo lenguaje, el ensamblador posee una sintaxis compuesta por un conjunto de instrucciones y sus operandos. Además, existen sentencias como directivas y comentarios que permiten completar la configuración del proyecto. Cada lenguaje ensamblador es diferente en su estructura, dependiendo de la marca y familia del microcontrolador. En ese sentido, un lenguaje ensamblador que funciona para el PIC18F no será equivalente para el PIC16F o para la familia dsPIC30F. Sin embargo, el conocimiento del lenguaje ensamblador para cualquier modelo de microcontrolador permite tener la experiencia necesaria para aprender rápidamente otro lenguaje ensamblador para otro modelo de microcontrolador.

Todo programa escrito en lenguaje ensamblador consiste en una secuencia de sentencias que mantienen el orden de ejecución de las operaciones por parte del procesador. Existen en total tres tipos de sentencias en un programa:

1. Instrucciones en lenguaje ensamblador. Estas son las instrucciones características del PIC18F. En total existen 77 instrucciones que el programador deberá conocer para poder sacar el máximo provecho al diseño digital con microcontroladores.
2. Directivas del lenguaje ensamblador. Las directivas son comandos que no generan un código máquina, pero que sí tienen una interpretación para el compilador que produce dicho código. Las directivas cumplen con funciones específicas, tales como definir bloques de instrucciones por compilar, establecer las direcciones de memoria de programa donde los bloques de instrucciones estarán almacenados, definir constantes y variables, establecer el final del programa, etc.
3. Comentarios. Los comentarios son parte fundamental de todo programa. Estos permiten colocar sentencias que ayudan al programador a entender la función de cada parte del programa. Los comentarios no modifican la funcionalidad del programa, y se recomienda que sean utilizados constantemente.

Este capítulo explica la sintaxis de las instrucciones del lenguaje ensamblador para el PIC18F. Para esto, se muestran algunos ejemplos donde se utilizan programas sencillos para demostrar la funcionalidad de algunas instrucciones. Posteriormente, se explican los modos de direccionamiento de la memoria de datos o SRAM para tener acceso a bloques de información. Luego, se analiza el uso del puntero de programa PC y el manejo del Puntero de tabla para la alteración del flujo de ejecución del programa y manipulación

de datos en la memoria FLASH. Posteriormente, se presentan las directivas del lenguaje ensamblador que el compilador reconoce y permiten mejorar la legibilidad del programa, así como facilitar la programación y definir la ubicación de bloques de instrucciones en la memoria FLASH del microcontrolador.

4.1 El conjunto de instrucciones

El código fuente o programa en lenguaje ensamblador se puede crear en cualquier editor de texto en formato ASCII (por ejemplo, el *Notepad* de *Windows*). Cada línea de instrucción puede contener cuatro campos diferentes:

1. Etiqueta
2. Mnemónico
3. Operando u operandos
4. Comentarios

La posición y el orden de estos cuatro campos es importante para evitar que el compilador MPASM genere advertencias al momento de generar el código máquina, y también para el mantenimiento de un orden que permita realizar un código legible.

Las etiquetas deben empezar en la primera columna del texto. Los mnemónicos, por otro lado, a partir de la segunda columna. Los operandos continúan luego del mnemónico y los comentarios siguen después de los operandos, mnemónicos o etiquetas, y se pueden ubicar en cualquier columna.

Es necesario el uso de espacios o dos puntos (:) para separar la etiqueta y el mnemónico, y el empleo de un espacio para separar el mnemónico del operando. Muchas veces el operando tiene más de un campo: en estas situaciones se utilizan comas para separar los diversos operandos.

Las etiquetas siempre se deben colocar en la primera columna. Una etiqueta sirve para identificar una posición de la memoria de programa donde inicia una determinada instrucción. El nombre que se le asigna a una etiqueta siempre debe comenzar con una letra del alfabeto, ya sea en minúscula o en mayúscula o con un guion bajo (_). El nombre de la etiqueta puede incluir caracteres numéricos a partir del segundo dígito. Cabe señalar que en todo el programa solo puede existir una única etiqueta con el mismo nombre. Cada etiqueta puede contener hasta 32 caracteres de largo y son sensibles a las mayúsculas y minúsculas. Normalmente, se utilizan dos puntos al final de la etiqueta para indicar que a continuación se presenta el mnemónico. Los dos puntos también se pueden reemplazar por un espacio en blanco. También es posible colocar el mnemónico en la siguiente línea debajo de la etiqueta.

Por ejemplo, las siguientes etiquetas son correctas:

```
1. INICIO      clrf TRISB
2. MAIN:       subwf 0x05
3. rutina:     addlw .30
4. V2  andwf 0x40
5. RETARDO_1SEG:
                decfsz 0x20,f
```

Por otro lado, las siguientes etiquetas presentan inconvenientes:

```
1.        MAIN:   incf 0x20          ; la etiqueta empieza en la sexta columna, esto
                                     ; generará una advertencia
2.  RETARDO-3SEG: movlw 0x15         ; Error debido al guion medio
3.  4MSEG_DELAY:  clrf 0x20          ; Error debido a que la etiqueta empieza con
                                     ; el dígito 4
```

Los mnemónicos vienen a ser los identificadores de las instrucciones en ensamblador o el especificador de las directivas, y siempre deben empezar desde la columna 2 para adelante. Cada mnemónico viene especificado en el conjunto de instrucciones para la arquitectura del procesador específico y también por el compilador utilizado. Los siguientes elementos son mnemónicos:

```
1.  Temporal EQU .10   ; EQU es una directiva del ensamblador
2.  goto INICIO        ; goto es un mnemónico
3.  movlw 0x20         ; movlw es un mnemónico
```

Por otro lado, los operandos son los campos que continúan al lado derecho del mnemónico. Algunas instrucciones no presentan operandos. Otras presentan un solo operando, y algunas, hasta dos o tres operandos. Los operandos se deben separar del mnemónico por uno o más espacios en blanco. Si se tiene más de un operando en la instrucción, estos se separarán con comas. En los siguientes ejemplos se muestran instrucciones con diferentes cantidades de operandos:

```
1.  movlw 0x30         ; 0x30 es el operando
2.  INICIO EQU 0       ; 0 es el operando
3.  movff 0x20,0x70    ; 0x20 y 0x70 son los operandos
4.  movf  0x50,W       ; 0x50 y W son los operandos
```

Finalmente, los comentarios son opcionales y se utilizan para documentar el funcionamiento de cada instrucción. Para iniciar un comentario se coloca un punto y coma (;). Todo lo que sigue después se considera un comentario y no forma parte del proceso de compilación. La herramienta MPASM del MPLAB X IDE muestra los comentarios en color verde para facilitar la ubicación de estos al programador. Los siguientes son ejemplos de comentarios:

```
1.  SUMA: addlw 0x50  ; Se suma el valor 0x50 al registro W
2.  incf  0x20        ; El contenido de la dirección 0x20 se incrementa en 1.
                      ;Rutina de retardo de 1 segundo (toda esta línea es un comentario)
```

Para iniciar ejecución de un programa se requiere un formato de inicialización básico en el cual se ejecuten las instrucciones rápidamente después del evento de Reset. Este formato se presenta a continuación, en el programa 4.1.

Programa 4.1. Formato de inicialización de un nuevo programa

```
cblock 0x00     ;Se define una directiva para la declaración de constantes comenzando
                ;desde el valor 0x00
  aux           ; aux es el alias al valor 0x00
  var           ; var es el alias al valor 0x01
endc            ; fin de la directiva cblock
org 0x000       ;Dirección 0x000 de la memoria de programa. Vector de Reset
goto MAIN
org 0x008       ;Vector de interrupción de alta prioridad. El salto a una rutina de servicio
                ; de interrupción de alta prioridad, debería incluirse en esta línea.
org 0x018       ;Vector de interrupción de baja prioridad. El salto a una rutina de servicio
                ;de interrupción de baja prioridad debería incluirse en esta línea.
org 0x020       ; En realidad cualquier dirección mayor a la 0x018 puede ir en esta
                ; directiva.
MAIN:           ; Aquí comienza el nuevo código del programa.

END             ; Directiva que indica el fin del programa. Cualquier instrucción debajo de
                ; esta directiva será ignorada por el compilador.
```

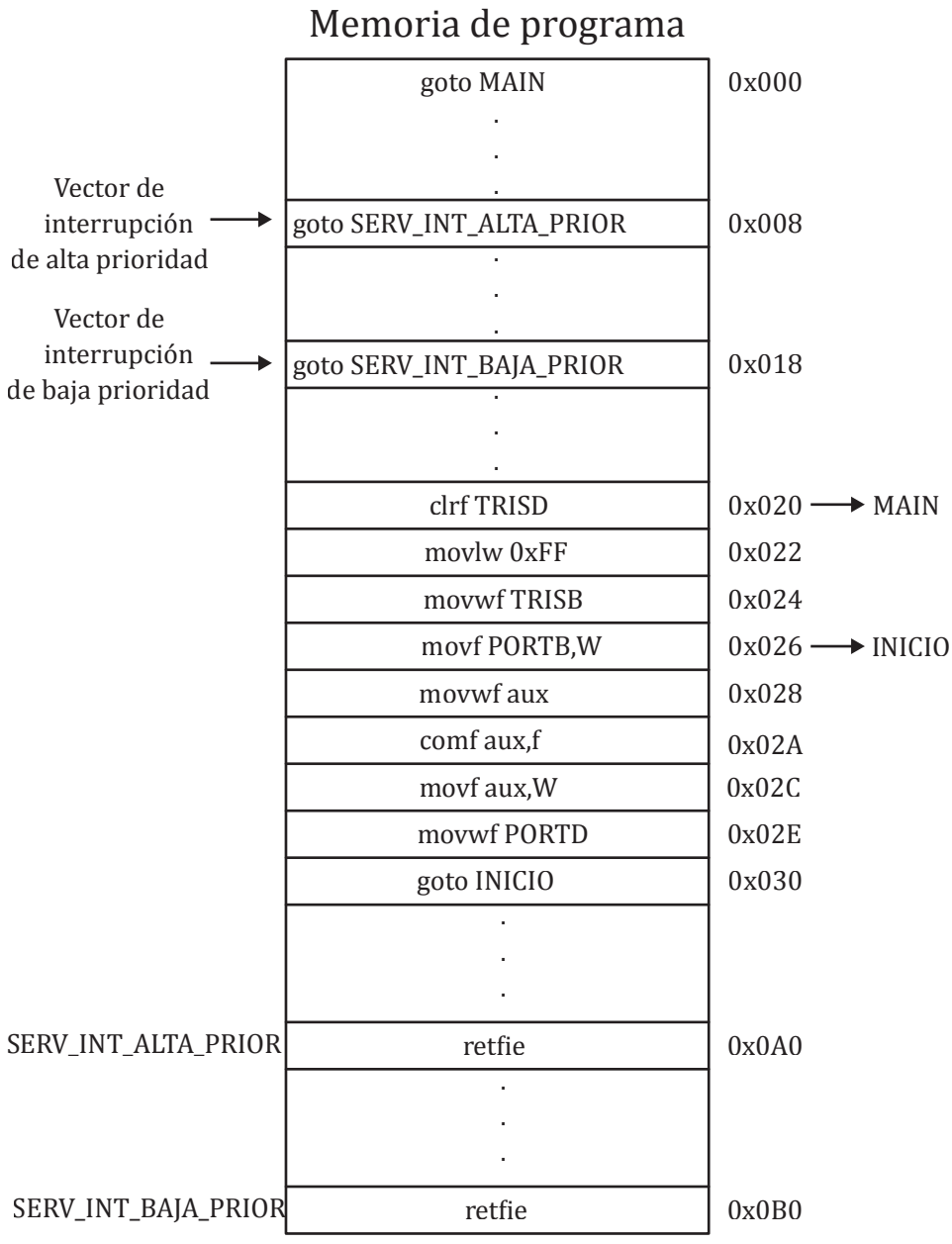
Por ejemplo, el programa 4.2, que se muestra a continuación, configura el Puerto B como entrada digital y el Puerto D como salida digital. Luego, el valor leído en el Puerto B se envía invertido continuamente al Puerto D.

Programa 4.2. Inversión del valor de entrada al Puerto B y presentación en el Puerto D

```
cblock 0x00
aux
endc
org 0x000
goto MAIN
org 0x008
goto SERV_INT_ALTA_PRIOR
org 0x018
goto  SERV_INT_BAJA_PRIOR
org 0x020
MAIN:
        clrf TRISD      ; TRISD = 0x00. Todos los pines del Puerto D son salidas digitales
        movlw 0xFF      ; W = 0xFF
        movwf TRISB     ; TRISB = W = 1111 1111. Todos los pines del Puerto B son entradas
                        ; digitales
INICIO:                 ; INICIO es una etiqueta que apunta a la dirección 0x23
        movf PORTB,W    ; W = PORTB, el valor de entrada al Puerto B se ingresa al registro
                        ; W
        movwf aux       ; aux = W. La dirección 0x00 de la SRAM contiene el valor leído en
                        ; el Puerto B
        comf aux,f      ; aux = complemento (aux). El valor contenido en la dirección 0x00
                        ; se niega
         movf aux,W     ; El valor negado guardado en la dirección 0x00 de la SRAM se
                        ;coloca en el registro W
        movwf PORTD     ; El valor en el registro W se envía el Puerto D
        goto   INICIO   ; Se retorna a la dirección 0x23 y se vuelven a ejecutar las
                        ;instrucciones desde este punto
org 0x0A0
 SERV_INT_ALTA_PRIOR:
        retfie
 org  0x0B0
SERV_INT_BAJA_PRIOR:
        retfie
 END
```

En el gráfico 4.1 se puede observar el esquema del programa presentado almacenado en la memoria FLASH del microcontrolador. Como se puede notar, la directiva ORG define la ubicación de las instrucciones en la memoria de programa. Cada instrucción ocupa una dirección completa de 16 bits. Como se explicó anteriormente, el Puntero de programa (PC) apunta a la dirección que será ejecutada y se incrementa de a dos, ya que su bit menos significativo es siempre 0.

Gráfico 4.1. Esquema del programa 4.1 almacenado en la memoria FLASH del microcontrolador



Elaboración propia.

El conjunto de instrucciones del microcontrolador se divide en función de su tipo de operación. Existen los siguientes tipos de instrucciones en el PIC18:

1. Orientadas a operaciones con bytes.
2. Orientadas a operaciones con bits.
3. Orientadas a operaciones con literales.
4. Orientadas a operaciones de control.

Además de conocer el mnemónico y los operandos de una instrucción, es importante saber el tiempo de duración que toma esta en ejecutarse. La mayoría de instrucciones demoran un ciclo de instrucción (cuatro periodos de reloj). Sin embargo, las instrucciones de salto tardan dos ciclos de instrucción, junto con la instrucción de manipulación de memoria (movff*)*, carga del registro FSRx y las de manejo del puntero de tabla.

4.1.1 Instrucciones orientadas a operaciones con bytes

1. **Instrucción: addwf f,d,a**

   Descripción:

   Esta instrucción suma el contenido del registro W con la dirección especificada en el operando f.

   Operandos:

   f -> dirección de la memoria de datos, cuyo contenido se va a sumar con el registro W.

   d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en el primer operando f.

   a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

   Duración: 1 ciclo de instrucción.

   Bits del registro STATUS afectados: C, DC, Z, OV, N

   Ejemplo:

```
movlw .25       ; Se carga W con el valor decimal .25
movwf 0x20      ; Se guarda el valor .25 almacenado en W en la dirección 0x20
movlw .10       ; Se carga W con el valor decimal .10
addwf 0x20,f,0  ; Se suma .10+.25 = .35 y se almacena en la dirección absoluta
                ; 0x20.
```

2. **Instrucción: addwfc f,d,a**

   Descripción:

   Esta instrucción suma el contenido del registro W con la dirección especificada en el operando f más el valor del bit *Carry* del registro STATUS.

   Operandos:

   f -> dirección de la memoria de datos, cuyo contenido se va a sumar con el registro W y con el bit *Carry*.

   d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

   a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

   Duración: 1 ciclo de instrucción.

   Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw .25        ; Se carga el registro W con el valor decimal .25
movwf 0x20       ; Se guarda el valor .25 en la dirección 0x20 de la SRAM
movlw .255       ; Se carga el registro W con el valor decimal .255
addwf 0x20,f,0   ; Se suma el valor .25 con .255 y se almacena en la dirección
                 ; 0x20. La suma genera un desbordamiento y el bit Carry se
                 ; pone a 1. El resultado es .24
movlw .10        ;Se carga el registro W con el valor decimal .10
addwfc 0x20,f,0 ;Se suma el valor del registro W con el valor almacenado en
                 ;dirección 0x20 más el bit Carry y se almacena en la dirección
                 ;0x20 el siguiente valor: .10 + .24 + .1 = .35.
```

3. **Instrucción: andwf f,d,a**

Descripción:

Esta instrucción realiza la operación lógica AND entre el contenido del registro W y la dirección especificada en el operando f.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido va a realizar la operación AND con el valor guardado en el registro W.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'10101101'    ;Se carga el registro W con el valor binario 1010 1101
movwf 0x20           ;Se guarda el valor binario 1010 1101 en la dirección 0x20
movlw b'00001111'    ;Se carga el registro W con el valor binario 0000 1111
andwf 0x20,f,0       ;Se realiza la operación AND entre el valor 0000 1111 y
                     ;1010 1101, lo cual da como resultado 0000 1101, el cual
                     ;se guarda en la dirección 0x20.
```

4. **Instrucción: clrf f,a**

Descripción:

Esta instrucción pone a la dirección de memoria especificada en f con el valor binario 00000000_2.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a mantener en el valor 0x00.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z

Ejemplo:

```
movlw 0xFF      ;Se carga en el registro W el valor 0xFF
movwf 0x40      ;Se guarda el valor 0xFF en la dirección 0x40 de la SRAM
clrf  0x40,0    ;Se carga el valor 00000000₂ en la dirección 0x40 sobre
                ;escribiendo el valor anterior.
```

5. **Instrucción: comf f,d,a**

Descripción:

Esta instrucción complementa (invierte) el valor binario contenido en la dirección destino f.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a complementar o negar.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'00110011'   ;Se carga el valor de W con el valor binario 0011 0011
movwf 0x40          ;Se guarda el valor binario 0011 0011 en la dirección
                    ;0x40
comf 0x40,f,0       ;Se invierte el contenido de la dirección 0x40
                    ;obteniéndose como resultado el valor binario 11001100₂,
                    ;almacenándose en la dirección 0x40 de la SRAM.
```

6. **Instrucción: cpfseq f,a**

Descripción:

Esta instrucción evalúa si el contenido de la dirección apuntada por f es igual al valor contenido en el registro de trabajo W. Si ambos valores son iguales, el PC se incrementa en 4 saltando una instrucción. En caso contrario, se ejecuta la instrucción siguiente, incrementándose el PC en 2.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a comparar con el contenido del registro W.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si no existe la igualdad. 2 ciclos de instrucción de existir la igualdad.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw b'11110000'     ;Cargar en el registro W el valor 11110000₂
movwf 0x40            ;Colocar el valor 11110000₂ en la dirección 0x40 de la
                      ;SRAM
comf  0x40,f,0        ;Negar el valor, generando el código 00001111₂ en la
                      ;dirección 0x40
cpfseq 0x40,0         ;Comparar el contenido de la dirección del registro 0x40
                      ;con W.
goto   DIFERENTE      ;Como son diferentes se ejecuta goto DIFERENTE
goto   IGUAL          ;Esta instrucción hubiera sido ejecutada si los valores
                      ;fueran sido iguales.
```

7. **Instrucción: cpfsgt f,a**

Descripción:

Esta instrucción evalúa si el contenido de la dirección apuntada por f es mayor al valor contenido en el registro de trabajo W. De ser verdadera la condición, el PC se incrementa en 4 saltando una instrucción. En caso contrario, se ejecuta la instrucción siguiente, incrementándose el PC en 2.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a comparar con el contenido del registro W.

d -> -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si el contenido de la dirección no es mayor a W. 2 ciclos en caso contrario.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw .18             ;Se carga el registro W con el valor decimal .18
movwf 0x20            ;Se guarda el valor .18 en la dirección 0x20
addwf 0x20,f,0        ;Se suma el valor .18 de la dirección 0x20 con el valor
                      ;almacenado en el registro W.
cpfsgt 0x20,0         ;Dado que la dirección 0x20 contiene el valor .36, es
                      ;mayor al valor .18 de W.
goto   MENOR_IGUAL    ;Esta instrucción no se ejecuta ya que el PC se ha
                      ;incrementado en 4.
goto   MAYOR          ;Esta instrucción es ejecutada debido al salto
                      ;producido por la comparación.
```

8. **Instrucción: cpfslt f,a**

Descripción:

Esta instrucción evalúa si el contenido de la dirección apuntada por f es menor al valor contenido en el registro de trabajo W. De ser verdadera la condición, el PC se incrementa en 4 saltando una instrucción. En caso contrario, se ejecuta la instrucción siguiente, incrementándose el PC en 2.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a comparar con el contenido del registro W.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si el contenido del registro f es mayor o igual al contenido del registro W. 2 ciclos de instrucción de ser menor.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw .18            ;Se carga el registro W con el valor decimal .18
movwf 0x20           ;Se almacena este valor en la dirección 0x20 de la SRAM
addwf 0x20,f,0       ;Se incrementa en .18 el valor de la dirección 0x20,
                     ;obteniéndose .36
cpfslt 0x20,0        ;Se compara el valor .36 con el .18 del registro W
goto   MAYOR_IGUAL ;Como el valor en memoria es mayor no hay salto y el
                     ;PC se incrementa en 2.
goto   MENOR         ;Esta condición no se ejecuta debido a que el valor en
                     ;el registro W es menor.
```

9. **Instrucción: decf f,d,a**

Descripción:

Esta instrucción resta en 1 el contenido de la dirección apuntada por f.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a restar en 1.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw .10        ;Se carga el valor decimal .10 al registro W
movwf 0x40       ;Se almacena el valor .10 en la dirección 0x40 de la SRAM
decf  0x40,f,0   ;Se resta en uno el valor en la dirección 0x40 y se almacena
                 ;el resultado en la misma dirección
decf  0x40,W,0   ;Se vuelve a restar el valor de la dirección 0x40 en uno, pero el
                 ;resultado se almacena en el registro W, con lo cual queda con
                 ;el valor decimal .8.
```

10. **Instrucción: decfsz f,d,a**

Descripción:

Esta instrucción resta en 1 el contenido de la dirección apuntada por f y evalúa si el resultado es 0. De ser así, salta una instrucción incrementando el PC en un factor de 4. De ser falsa la condición, se ejecuta la siguiente instrucción.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a restar en un factor de 1 el resultado se va a comparar con el valor 0.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si el contenido el resultado del decremento es diferente de 0.2 ciclos de instrucción de ser 0x00 la resta.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .10        ;Se carga el registro W con el valor decimal .10
        movwf 0x20       ;Se almacena el valor .10 en la dirección 0x20 de la SRAM
LAZO:
        decfsz 0x20,f,0  ;Se resta en 1 el contenido de la dirección 0x20 y se compara
                         ;con 0
        goto   LAZO      ;Si la resta es un número diferente de 0 se ejecuta esta
                         ;instrucción
CONTINUA:                ;Después de 10 restas, el valor de la dirección 0x20 es 0 y por
        movf   0x20,W,0  ;tanto se ejecuta esta instrucción donde el contenido de la
                         ;dirección 0x20 se carga en W (valor 0).
```

11. **Instrucción: dcfsnz f,d,a**

Descripción:

Esta instrucción resta en 1 el contenido de la dirección apuntada por f y evalúa si el resultado es diferente de 0. De ser verdadera esta afirmación, salta una instrucción incrementando el PC en un factor de 4. De ser 0 el valor del registro, se ejecuta la siguiente instrucción.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a restar en un factor de 1 el resultado se va a comparar con el valor de 0.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 2 ciclos de instrucción si el contenido del resultado del decremento es diferente de 0. 1 ciclo de instrucción de ser 0 la resta.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .3          ;Cargar el valor decimal .3 en el registro W.
        movwf 0x20        ;Cargar el valor .3 en la dirección 0x20 de la SRAM.
LAZO:
        dcfsnz 0x20,f,0   ;Se resta en uno el valor en la dirección 0x20 y se compara con
                          ;0. La resta da .2
        goto   LAZO       ;Entonces se salta una instrucción y no se ejecuta goto LAZO
CONTINUA:
        movf   0x20,W,0   ;Se carga el valor 0x20 en el registro W, con lo que W toma el
                          ;valor de .2.
```

12. **Instrucción: incf f,d,a**

Descripción:

Esta instrucción incrementa en 1 el contenido de la dirección apuntada por f.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a incrementar en un factor de 1.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

*Bits del registro STATUS afectados:*C, DC, Z, OV, N

Ejemplo:

```
movlw .4          ;Cargar el valor decimal .4 en el registro W
movwf 0x20        ;Cargar el valor decimal .4 en la dirección 0x20 de la SRAM
incf 0x20,f,0     ;Incrementar en 1 el valor de la dirección 0x20, ahora contiene el
                  ;valor .5
incf 0x20,W,0     ;Incrementar en 1 el valor de la dirección 0x20 y se guarda en el
                  ;registro W el valor .6
```

13. **Instrucción: incfsz f,d,a**

Descripción:

Se incrementa en 1 el valor contenido en la dirección especificada por f. Se compara el resultado con 0 y se salta una instrucción de ser verdadera esta condición. De ser falsa, no habrá salto.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido se va a incrementar en un factor de 1 y se va a comparar con el valor 0.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si el incremento no da como resultado 0. 2 ciclos de instrucción si el incremento da como resultado 0.
Bits del registro STATUS afectados: Ninguno
Ejemplo:

```
        movlw  .255      ;Se carga el valor 11111111₂ en el registro W
        movwf  0x20      ;Ese mismo valor se carga en la dirección 0x20 de la SRAM
        incfsz 0x20,f,0  ;Se incrementa en uno el contenido de la dirección 0x20
                         ;con lo cual es 0.
        movlw  'A'       ;Como la condición es verdadera, esta instrucción no se ejecuta y
                         ;el registro W mantiene su valor inicial en 11111111₂
LAZO:
        goto LAZO        ;Se ejecuta esta instrucción debido al salto ocasionado por la
                         ;condición verdadera.
```

14. **Instrucción: infsnz f,d,a**

Descripción:
Se incrementa en 1 el valor contenido en la dirección especificada por f. Se compara el resultado con 0 y salta una instrucción de no ser verdadera esta condición. De ser 0 el resultado, no habrá salto.
Operandos:
f -> dirección de la memoria de datos, cuyo contenido se va a incrementar en un factor de 1 y se va a comparar con el valor 0.
d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.
a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).
Duración: 1 ciclo de instrucción si el incremento da como resultado 0. 2 ciclos de instrucción si el incremento da como resultado un valor diferente de 0.
Bits del registro STATUS afectados: Ninguno
Ejemplo:

```
        movlw  .255      ;Cargar el registro W con el valor 1111 1111₂
        movwf  0x20      ;Mover el valor 1111 1111₂ a la dirección 0x20 de la SRAM
        infsnz 0x20,f,0  ;Incrementar en 1 el valor de la dirección 0x20 y saltar si no es 0
                         ;el resultado.
        incf   0x20,f,0  ;Como el resultado es 0, entonces esta instrucción se ejecuta y el
                         ;nuevo resultado es 1
        infsnz 0x20,f,0  ;Se incrementa en 1 el valor de la dirección 0x20 y al ser .2
                         ;(diferente de 0) se salta una instrucción.
        clrf   0x20      ;Por lo tanto clrf 0x20, no se ejecuta.
LAZO:
        goto LAZO        ;Esta instrucción se ejecuta luego de la segunda instrucción
                         ;infsnz.
```

15. **Instrucción: iorwf f,d,a**

Descripción:

Instrucción que ejecuta la operación lógica OR inclusiva entre una dirección de memoria SRAM y el contenido del registro W.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido va a realizar la operación lógica OR inclusiva con el valor del registro W.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z,N

Ejemplo:

```
movlw  b'00010001'    ;Se carga en el registro W el valor binario 00010001₂
movwf  0x40           ;Se coloca el valor 00010001₂ en la dirección 0x40 de la
                      ;SRAM
movlw  b'10000101'    ;Se carga el registro W con el valor binario 10000101₂
iorwf  0x40,f,0       ;Se realiza la operación OR entre W y el valor contenido
                      ;en la dirección 0x40. El resultado es 10010101₂ y se
                      ;almacena en la dirección 0x40.
```

16. **Instrucción: movf f,d,a**

Descripción:

Esta instrucción copia el contenido de la dirección de memoria SRAM especificada en f al registro W.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido va a cargarse en el registro W.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z,N

Ejemplo:

```
movlw  0x64       ;Se carga el registro W con el valor hexadecimal 0x64
movwf  0x40       ;Se guarda el valor 0x64 en la dirección 0x40 de la SRAM
movlw  0x01       ;Se carga el registro W con el valor 0x01
movf   0x40,W,0 ;Se carga el registro W con el valor contenido en la
                  ;dirección 0x40 de la SRAM: 0x64.
```

17. **Instrucción: movff fs,fd**

Descripción:

Esta instrucción copia el contenido de la dirección especificada en el campo fs a la dirección destino especificada en el campo fd.

Operandos:

fs -> dirección fuente del contenido de los datos. Este operando puede ser de 12 bits de longitud.

fd -> dirección destino. Este operando puede ser de 12 bits de longitud.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw  'A'          ;Se carga el valor ASCII 'A' (0x41) en el registro W
movwf  0x40         ;Se copia el contenido del registro W a la dirección 0x40 de la
                    ;SRAM
movff  0x40,0x20 ;El contenido de la dirección 0x40 se carga en la dirección
                    ;0x20
```

18. **Instrucción: movwf f,a**

Descripción:

Esta instrucción copia el contenido del registro W en la dirección especificada en el campo f.

Operandos:

f -> dirección de la memoria de datos, cuyo contenido va a tomar el valor almacenado en el registro W.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw  'B'        ;Se carga el valor ASCII 'B' (0x42) en el registro W
movwf  0x40,0     ;Se carga el valor ASCII 'B' del registro W a la dirección 0x40
                  ;de la SRAM.
```

19. **Instrucción: mulwf f,a**

Descripción:

Esta instrucción multiplica el valor del registro W con el valor almacenado en la dirección especificada en el campo f. El resultado, el cual puede ser una cifra mayor a 8 bits, ya que la multiplicación puede superar el valor de .255, se almacena en dos registros: PRODH (que contiene los 8 bits con la parte alta del resultado) y PRODL (que contiene los 8 bits menos significativos del resultado).

Operandos:

f -> dirección de la memoria de datos, cuyo contenido es uno de los operandos que se multiplicará con el valor almacenado en el registro W.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw  .128          ;Se carga el registro W con el valor decimal .128
movwf  0x20,0        ;Se ingresa el valor .128 a la dirección 0x20 de la SRAM
movlw  .8            ;Se carga el registro W con el valor decimal .8
mulwf  0x20,0        ;Se multiplica el valor .8 del registro W con el .128 de la
                     ;dirección 0x20
movff  PRODL,0x40    ;El resultado es 1024 = 0x0400, en donde PRODL = 0x00
                     ;que se carga en la dirección 0x40 y
movff  PRODH,0x60    ;PRODH = 0x04 que se carga en la
                     ;dirección 0x60 de la SRAM.
```

20. **Instrucción: negf f,a**

Descripción:

Esta instrucción obtiene el complemento a dos de una cifra almacenada en una dirección de la memoria SRAM especificada por f.

Operandos:

f -> dirección de la memoria de datos a cuyo contenido se le calculará su valor complemento a 2.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw  b'11111101' ;Se carga en el registro W el valor binario 1111 1101₂ que
                   ;equivale a -3 en formato complemento a 2.
movwf  0x20,0      ;Se guarda el valor 1111 1101₂ en la dirección 0x20 de la
                   ;SRAM.
negf   0x20,0      ;Se obtiene el complemento a 2 y el resultado genera el
                   ;valor .3 .
```

21. **Instrucción: rlcf f,d,a**

Descripción:

Esta instrucción rota un bit a la izquierda el valor especificado por la dirección de f. El bit LSB después del corrimiento toma el valor del bit *Carry* del registro STATUS. Una vez usado el bit *Carry* este se pone en '0'.

Operandos:

f ->dirección de la memoria de datos a cuyo registro se le realizará la rotación a la izquierda

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, Z, N

Ejemplo:

```
movlw  0xFF          ;Se carga el registro W con el valor 1111 1111(2)
movwf  0x20          ;Se ingresa el valor del registro W a la dirección 0x20 de la
                     ;SRAM
incf   0x20,0        ;Se incrementa en 1 el valor de la dirección 0x20, con lo cual el
                     ;bit Carry es '1'.
movlw  b'00000001'   ;Se coloca el valor binario 0000 0001 en el registro W
movwf  0x20,0        ;Se guarda el valor 0000 0001 en la dirección 0x20
rlcf   0x20,f,0      ;Se rota a la izquierda una posición el valor de la dirección 0x20,
                     ;pero, su bit LSB toma el valor del Carry, quedando la cifra de la
                     ;siguiente manera: 0000 0011(2)
rlcf   0x20,f,0      ;Se vuelve a rotar la cifra una posición a la izquierda quedando:
                     ;0000 0110(2).
```

22. **Instrucción: rlncf f,d,a**

Descripción:

Esta instrucción rota un bit a la izquierda el valor especificado por la dirección dada en f. La rotación no utiliza el bit *Carry* del registro STATUS. Cuando se realiza la rotación a la izquierda el bit MSB, toma el valor del bit LSB en el registro.

Operandos:

f -> dirección de la memoria de datos a la cual se le realizará la rotación de un bit.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw  b'01000001'   ;Se carga en el registro W el valor 0100 0001(2)
movwf  0x20,0        ;Se carga el valor binario de W en la dirección 0x20 de la
                     ;SRAM.
rlncf  0x20,f,0      ;Se rota un bit a la izquierda el valor de la posición 0x20
                     ;quedando 1000 0010(2)
rlncf  0x20,f,0      ;Se vuelve a rotar un bit a la izquierda el mismo registro
                     ;quedando 00000 0101(2)
rlncf  0x20,f,0      ;Otra vez se realiza la misma operación sobre el registro
                     ;quedando 0000 1010(2).
```

23. **Instrucción: rrcf f,d,a**

Descripción:

Esta instrucción rota un bit a la derecha el valor especificado por la dirección de f. El bit MSB después del corrimiento toma el valor del *Carry* del registro STATUS. Una vez usado el bit *Carry*, este se pone a 0.

Operandos:

f ->dirección de la memoria de datos a cuyo registro se le realizará la rotación a la derecha.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, Z, N

Ejemplo:

```
movlw  0xFF          ;Se carga el registro W con el valor 11111111₂
movwf  0x20          ;Se ingresa el valor del registro W a la dirección 0x20 de la
                     ;SRAM
incf   0x20,0        ;Se incrementa en 1 el valor de la dirección 0x20, con lo cual el
                     ;bit Carry es '1'.
movlw  b'01000000'   ;Se coloca el valor binario 0100 0000₂ en el registro W
movwf  0x20,0        ;Se guarda el valor 0100 0000₂ en la dirección 0x20
rrcf   0x20,f,0      ;Se rota a la derecha una posición el valor de la dirección
                     ;0x20, pero, su bit MSB toma el valor del bit Carry que es
                     ;'1', es decir, el resultado es 10100000₂
rrcf   0x20,f,0      ;Se vuelve a rotar la cifra una posición a la derecha
                     ;quedando: 01010000₂.
```

24. **Instrucción: rrncff,d,a**

Descripción:

Esta instrucción rota un bit a la derecha el valor especificado por la dirección dada en f. La rotación no utiliza el bit *Carry* del registro STATUS. Cuando se realiza la rotación a la derecha, el bit LSB toma el valor del bit MSB en el registro.

Operandos:

f ->dirección de la memoria de datos a cuyo registro se le realizará la rotación a la derecha.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

*Duración:*1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'01000010'    ;Se carga el valor binario 0100 0010 en el registro W.
movwf 0x20,0         ;El contenido del registro W se carga en la dirección 0x20
                     ;de la SRAM.
rrncf 0x20,f,0       ;Se realiza la rotación del valor contenido en la dirección ;
                     ;0x20 quedando 00100001₂.
rrncf 0x20,f,0       ;Se realiza la misma operación sobre el mismo registro
                     ;quedando 10010000₂.
rrncf 0x20,f,0       ;Nuevamente se repite el proceso sobre el mismo registro
                     ;quedando 01001000₂.
```

25. **Instrucción: setf f,a**

Descripción:

Esta instrucción pone todos los bits en '1' del registro apuntado por la dirección f.

Operandos:

f -> dirección del registro de la SRAM que tendrá todos sus bits a nivel alto después de la instrucción.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

*Duración:*1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
clrf 0x20,0      ;El registro de la dirección 0x20 de la SRAM contiene el valor
                 ;00000000₂.
setf 0x20,0      ;El registro de la dirección 0x20 de la SRAM contiene el valor
                 ;11111111₂.
```

26. **Instrucción: subfwb f,d,a**

Descripción:

Esta instrucción le resta el valor contenido de la dirección de la SRAM apuntada por f al valor en el registro W y, además, resta el bit *Borrow* (Si el bit *Carry* es '0', entonces el bit *Borrow* es '1').

Operandos:

f ->Dirección del registro de la SRAM que se restará al valor contenido en el registro W junto con el bit *Borrow*.

d -> Destino del resultado. Puede tomar el valor '0' o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw .20        ;Se carga el valor decimal .20 en el registro W.
movwf 0x20,0     ;Se guarda el valor decimal. 20 en la dirección 0x20
movlw .30        ;Se carga el valor decimal .30 en el registro W
clrf STATUS,0    ;Se pone el bit Carry del registro STATUS a nivel bajo (el
                 ;bit Borrow = '1')
subfwb 0x20,f,0 ;Se realiza la resta .30-.20-.1 = .9. El resultado se guarda
                 ;en el registro de la dirección 0x20 de la SRAM.
```

27. **Instrucción: subwf f,d,a**

Descripción:

Esta instrucción resta el valor del registro W (sustraendo) al registro apuntado por la dirección f de la SRAM (minuendo).

Operandos:

f -> dirección de la memoria de datos, el cual será el minuendo de la operación.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw .100       ;Se carga el valor decimal .100 en el registro W.
movwf 0x40,0     ;El valor .100 se guarda en el registro que ocupa la dirección
                 ;0x40 de la SRAM.
movlw .50        ;Se carga el valor decimal .50 en el registro W
subwf 0x40,W,0  ;Se obtiene la resta W = f – W = .100 - .50 = .50. El
                 ;resultado se guarda en el registro W
```

28. **Instrucción: subwfb f,d,a**

Descripción:

Esta instrucción resta el valor del registro W (primer sustraendo) y el valor del bit *Borrow* (segundo sustraendo) del valor apuntado por la dirección f de la SRAM.

Operandos:

f -> dirección de la SRAM, cuyo contenido será el minuendo de la operación.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: C, DC, Z, OV, N

Ejemplo:

```
movlw .200          ;Se carga el valor decimal .200 al registro W
movwf 0x20,0        ;El valor .200 se ingresa al registro apuntado por la dirección
                    ;0x20 de la SRAM
movlw .100          ;Se carga el valor decimal .100 al registro W
clrf  STATUS,0      ;El bit Carry se pone a '0' lógico y por tanto el bit
                    ;Borrow es '1' lógico.
subwfb 0x20,f,0     ;Se realiza la resta y el resultado se guarda en el registro
                    ;apuntado por la dirección 0x20 de la SRAM. El valor
                    ;almacenado es .200 - .100 - .1 = .99.
```

29. **Instrucción: swap f,d,a**

Descripción:

Esta instrucción intercambia los valores de los *nibbles* del registro apuntado por la dirección f de la memoria de datos.

Operandos:

f -> es la dirección de la memoria de datos que apunta al registro que contiene el valor binario sobre el cual se realizará la operación.

d -> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw 0x57          ;Se carga el valor hexadecimal 0x57 en el registro W
movwf 0x20,0        ;El valor 0x57 se ingresa al registro apuntado por la dirección
                    ;0x20 de la SRAM
swapf 0x20,f,0      ;Se intercambian los nibbles del registro apuntado por la
                    ;dirección 0x20 de la SRAM quedando el resultado 0x75.
```

30. **Instrucción: tstfsz f,a**

Descripción:

Esta instrucción compara el valor apuntado por la dirección de la memoria de datos dada por f con 0. Si la igualdad es verdadera, el PC se incrementa en 4 saltando una instrucción. En caso contrario, se ejecuta la siguiente instrucción.

Operandos:

f -> dirección de la memoria de datos que contiene el valor que se va a comparar con 0.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción de ser diferente de 0. 2 ciclos si la dirección de memoria contiene un 0.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
clrf  0x20,0        ;El valor apuntado por la dirección 0x20 de la SRAM se pone en
                    ;0.
tstfsz 0x20,0       ;Se compara el valor de la dirección 0x20 con 0 y salta una
                    ;instrucción por ser verdadero.
setf  0x20,0        ;Se ponen todos los bits a '1' lógico, pero, esta instrucción no se
                    ;ejecuta debido al resultado anterior.
incf  0x20,f,0      ;Se incrementa en 1 el valor de la dirección 0x20 quedando el
                    ;resultado en el valor .1.
```

31. **Instrucción: xorwf f,d,a**

Descripción:

Esta instrucción realiza la operación XOR entre el valor contenido en el registro W y el registro apuntado por la dirección de la memoria de datos f.

Operandos:

f -> dirección de la memoria de datos que apunta al registro cuyo contenido será un operando de la operación XOR.

d-> destino del resultado. Puede tomar el valor 0 o W para seleccionar el registro W. Si toma el valor 1 o f, selecciona la dirección de la memoria SRAM especificada en f.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'11000011'   ;Se carga el valor binario 11000011₂ en el registro W
movwf 0x20,0        ;El valor 11000011₂ se guarda en el registro apuntado por
                    ;la dirección 0x20
movlw b'11000001'   ;Se carga el valor binario 11000001₂ en el registro W
xorwf 0x20,W,0      ;Se realiza la operación XOR. El resultado se guarda en el
                    ;registro W siendo el valor binario 00000010₂.
```

4.1.2 Instrucciones orientadas a operaciones con bits

1. **Instrucción: bcf f,b,a**

Descripción: Esta instrucción pone en nivel '0' el bit b (que puede tomar cualquier valor entre 0 y 7) del registro especificado por la dirección f de la memoria de datos.

Operandos:

f -> dirección de la memoria de datos donde uno de sus 8 bits va a tomar el nivel '0' lógico.

b -> posición del bit que va a ser puesto a nivel '0'. El valor 0 especifica el bit LSB, y el valor 7 (máximo valor), el bit MSB.

a -> dirección absoluta ('0') o relativa al banco de memoria SRAM ('1').

Duración: 1 ciclo de instrucción.
Bits del registro STATUS afectados: Ninguno
Ejemplo:

```
setf  0x20,0        ;El registro de la dirección 0x20 de la SRAM toma el valor
                    ;11111111₂.
bcf   0x20,7,0      ;El bit 7 del registro en mención se pone a 0 lógico, resultando el
                    ;valor 01111111₂.
bcf   0x20,0,0      ;El bit 0 del registro en mención se pone a 0 lógico, resultando el
                    ;valor 01111110₂.
```

2. **Instrucción: bsf f,b,a**
Descripción:
Esta instrucción pone en nivel '1' el bit b (que puede tomar cualquier valor entre 0 y 7) del registro especificado por la dirección f de la memoria de datos.
Operandos:
f -> dirección de la memoria de datos donde uno de sus 8 bits va a tomar el nivel '1'.
b -> posición del bit que va a ser puesto a nivel '1'. El valor 0 especifica el bit LSB, y el valor 7 (máximo valor), el bit MSB.
a -> dirección absoluta ('0') o relativa al banco de memoria SRAM ('1').
Duración: 1 ciclo de instrucción.
Bits del registro STATUS afectados: Ninguno
Ejemplo:

```
clrf  0x20,0        ;El registro de la dirección 0x20 de la SRAM toma el valor
                    ;00000000₂.
bsf   0x20,7,0      ;El bit 7 del registro en mención se pone a 0 lógico, resultando el
                    ;valor 10000000₂.
bsf   0x20,0,0      ;El bit 0 del registro en mención se pone a 0 lógico, resultando el
                    ;valor 10000001₂.
```

3. **Instrucción: btfsc f,b,a**
Descripción:
Esta instrucción analiza el bit b del registro apuntado por la dirección de la memoria de datos especificado en f. Si el bit es '0', se salta una instrucción incrementando el PC en 4. En caso que el bit sea '1', se ejecuta la siguiente instrucción.
Operandos:
f -> dirección de la memoria de datos donde uno de sus 8 bits va a ser analizado.
b -> posición del bit que va a ser analizado. El valor 0 especifica el bit LSB, y el valor 7 (máximo valor), el bit MSB.
a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).
Duración: 1 ciclo de instrucción si la condición del bit es falsa. 2 ciclos si es verdadera.
Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw b'10101010'     ;Se carga el registro W con el valor binario 1010 1010(2).
movwf 0x40,0          ;El valor del registro W se carga al registro apuntado por
                      ;la dirección 0x40 de la SRAM.
btfsc 0x40,2,0        ;Se analiza el segundo bit del registro de la dirección
                      ;0x40 y si es '0' salta 1 instrucción.
clrf  0x40,0          ;Dado que el bit 2 es 0, la instrucción clrf no se ejecuta
incf  0x40,0          ;Se ejecuta esta instrucción y el valor almacenado en la
                      ;dirección 0x40 es 1010 1011(2).
```

4. **Instrucción: btfss f,b,a**

Descripción:

Esta instrucción analiza el bit b del registro apuntado por la dirección de la memoria de datos especificado en f. Si el bit es 1_2, se salta una instrucción incrementando el PC en 4. En caso de que el bit sea '0', se ejecuta la siguiente instrucción.

Operandos:

f -> dirección de la memoria de datos donde uno de sus 8 bits va a ser analizado.

b -> posición del bit que va a ser analizado. El valor 0 especifica el bit LSB, y el valor 7 (máximo valor), el bit MSB.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción si la condición del bit es falsa. 2 ciclos si es verdadera.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw b'10101010'     ;Se carga el registro W con el valor binario 1010 1010
movwf 0x40,0          ;El valor del registro W se carga al registro apuntado por
                      ;la dirección 0x40 de la SRAM.
btfss 0x40,2,0        ;Se analiza el segundo bit del registro de la dirección
                      ;0x40 y si es '1'  salta 1 instrucción.
clrf  0x40,0          ;Dado que el bit 2 es 0, no hay salto y la instrucción clrf se
                      ;ejecuta generando un 0x00.
incf  0x40,0          ;Se incrementa el valor de la dirección 0x40, quedando
                      ;como resultado 0x01.
```

5. **Instrucción: btg f,b,a**

Descripción:

Esta instrucción invierte el estado del bit b, correspondiente al registro de la memoria de datos apuntado por la dirección f.

Operandos:

f -> dirección de la memoria de datos donde uno de sus 8 bits va a tomar el nivel '1'.

b -> posición del bit que va a tener inversión de nivel. El valor 0 especifica el bit LSB, y el valor 7 (máximo valor), el bit MSB.

a -> dirección absoluta (0) o relativa al banco de memoria SRAM (1).

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw b'10101010'    ;Se carga el registro W con el valor binario 10101010₂
movwf 0x40,0         ;El valor 1010 1010 ingresa al registro apuntado por la
                     ;dirección 0x40.
btg   0x40,7,0       ;Se invierte el bit 7 del registro, quedando como resultado
                     ;00101010₂.
```

4.1.3 Instrucciones orientadas a operaciones con literales

1. **Instrucción: addlw k**

   Descripción:

   Esta instrucción suma el valor literal k al contenido del registro W. El resultado se guarda en el registro de trabajo W.

   Operandos:

   k -> Valor literal de 8 bits, el cual será inmediatamente sumado al contenido del registro W.

   Duración: 1 ciclo de instrucción

   Bits del registro STATUS afectados: C, DC, Z, OV, N

   Ejemplo:

```
movlw .15      ;Se carga el registro W con el valor decimal.15.
addlw .45      ;Se suma al valor .15 el valor literal decimal .45 el cual da como
               ;resultado .60.
movwf 0x40,0   ;Se guarda en la dirección 0x40 de la SRAM el valor decimal .60
```

2. **Instrucción: andlw k**

   Descripción:

   Esta instrucción realiza la operación lógica AND entre el valor literal k y el contenido del registro W. El resultado se guarda en el registro de trabajo W.

   Operandos:

   k -> Valor literal de 8 bits, el cual será un operando de la operación lógica AND con el contenido del registro W.

   Duración: 1 ciclo de instrucción.

   Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'11110011'    ;Se carga en el registro W el valor binario 11110011(2).
andlw b'00001111'    ;Se realiza la operación AND con  el valor 00001111(2)
                     ;obteniéndose como
movwf 0x40,0         ;resultado 00000011(2), el cual se guarda en la dirección
                     ;0x40 de la SRAM.
```

3. **Instrucción: iorlw k**

Descripción:

Esta instrucción realiza la operación lógica OR entre el valor literal k y el contenido del registro W. El resultado se guarda en el registro de trabajo W.

Operandos:

k -> Valor literal de 8 bits, el cual será un operando de la operación lógica OR con el contenido del registro W.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Z, N

Ejemplo:

```
movlw b'00000011'  ;Se carga en el registro W el valor binario 0000 0011(2).
iorlw b'11000011'  ;Se realiza la operación lógica OR con el valor binario
                   ;1100 0011 y como resultado se obtiene 1100 0011(2)
movwf 0x40,0       ;valor que se guarda en la dirección 0x40 de la SRAM.
```

4. **Instrucción: lfsr f,k**

Descripción:

Esta instrucción carga el valor literal k en cualquiera de los registros FSR0, FSR1 o FSR2 usados para los modos de direccionamiento indirecto.

Operandos:

f -> puede ser 0 (FSR0), 1 (FSR1) o 2 (FSR2). Indica cuál de los registros FSRx (donde x = 0, 1 o 2) será cargado con el valor k.

k -> Es un valor literal de 12 bits que indica la dirección de la memoria SRAM a la cual apuntará el registro FSRx.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
lfsr  FSR0,0x300     ;Se carga el registro FSR0 con el valor 0x300, apuntando
                     ;a dicha dirección de la SRAM.
```

5. **Instrucción: movlb k**

Descripción:

Esta instrucción carga el valor literal k al registro BSR, el cual apunta a uno de los 8 bancos de la memoria de datos. Esto se utiliza para los modos de acceso indirecto de memoria.

Operandos:

k -> Valor literal de 8 bits que especifica el banco a acceder, que es almacenado en el registro BSR. No debería superar el valor de .15.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlb .2        ;Se carga el valor decimal .2 en el registro BSR. Se selecciona
                ;el Banco 2 (dirección de inicio 0x200. Se carga el valor
movlw 0x88      ;hexadecimal 0x88 en el registro W.
movwf 0x10,1    ;Se carga el valor de W a la dirección 0x10 + 0x200 = 0x210 de
                ;la memoria SRAM.
```

6. **Instrucción: movlw k**

Descripción:

Esta instrucción carga el valor literal k directamente en el registro W.

Operandos:

k -> Valor literal de 8 bits que será cargado en el registro W.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw .9        ;El registro W se carga con el valor 00001001₂.
```

7. **Instrucción: mullw k**

Descripción:

Esta instrucción multiplica el valor del registro W con el literal k. El resultado de la multiplicación se almacena en los registros PRODL (los 8 bits de la parte baja del resultado) y PRODH (los 8 bits de la parte alta del resultado).

Operandos:

k -> Valor literal de 8 bits que será multiplicado con el valor del registro W.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
movlw .80                ;Se carga el registro W con el valor decimal .80.
mullw .100               ;Se multiplica .80x.100, lo cual da 8000 o 0x1F40 en
                         ;hexadecimal.
movff PRODL,0x20         ;Se carga la dirección 0x20 de la SRAM con el
                         ;valor 0x40.
movff PRODH,0x21         ;Se carga la dirección 0x21 de la SRAM con el
                         ;valor 0x1F.
```

8. **Instrucción: retlw k**

Descripción:

Esta instrucción permite retornar a la posición siguiente después de la ejecución de la última instrucción *call*. El retorno se realiza con el valor del registro W conteniendo el literal k

Operandos:

k -> Valor literal de 8 bits que será cargado al registro W en el momento del retorno de la instrucción *call*.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .50            ;Se carga el valor decimal .50 al registro W
        movwf 0x20,0         ;Se carga el valor decimal .50 en el registro apuntado por la
                             ;dirección 0x20 de la SRAM
        call  RUTINA         ;Se salta a la etiqueta RUTINA
        addwf 0x20,W,0       ;Se retorna de RUTINA y se suma el valor de W (.5) al
                             ;valor .50 de la dirección 0x20
        goto FIN             ;obteniéndose como resultado .55 almacenado en el registro W.
                             ;Luego, se salta a la etiqueta FIN
RUTINA:
        retlw .5             ;Se retorna a la siguiente instrucción luego del call, pero, con el
                             ;valor de W = .5.
        FIN:
```

9. **Instrucción: sublw k**

   Descripción:

   Esta instrucción resta el valor del registro W (sustraendo) del valor literal k (minuendo) y almacena el resultado en el registro W.

   Operandos:

   k -> Valor literal de 8 bits el cual será el minuendo de la operación.

   Duración: 1 ciclo de instrucción.

   Bits del registro STATUS afectados: C, DC, Z, OV, N

   Ejemplo:

```
movlw .50       ;Se carga el valor decimal .50 en el registro W
sublw .120      ;W = .120-.50 = .70.
```

10. **Instrucción: xorlw k**

    Descripción:

    Esta instrucción realiza la operación XOR entre el contenido del registro W y el literal k. El resultado se almacena en el registro W.

    Operandos:

    k -> Valor literal de 8 bits que será uno de los operandos de la instrucción XOR.

    Duración: 1 ciclo de instrucción.

    Bits del registro STATUS afectados: Z, N

    Ejemplo:

```
movlw b'01010101'    ;Se carga el registro W con el valor binario 01010101₂,
                     ;luego se realiza la
xorlw b'10101010'    ;operación XOR con 1010 1010, quedando como
                     ;resultado 11111111₂ en el registro W.
```

4.1.4 Instrucciones orientadas a operaciones de control

1. **Instrucción: bc n**

   Descripción:

   Esta instrucción salta a la dirección de la memoria de programa especificada en n (el PC toma el valor n) si es que el bit *Carry* del registro STATUS es '1'. En caso contrario, no se produce ningún salto.

   Operandos:

   n -> Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

   Duración: 2 ciclos de instrucción si se produce el salto. 1 ciclo de instrucción si el bit *Carry* es '0'.

   Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        org 0x00        ;La directiva ORG establece que la siguiente instrucción será
                        ;establecida en la dirección
        goto MAIN       ;0x00 de la memoria de programa: goto MAIN
        org 0x20        ;El inicio del programa se establece en la dirección 0x20 de la
                        ;memoria de programa
MAIN:
        movlw .255      ;Se carga el valor decimal .255 en el registro W
        addlw .1        ;Se incrementa el valor .255 en .1, produciéndose un acarreo
                        ;(Carry = '1')
        bc   0x28       ;Como el Carry = '1', se salta a la dirección 0x28 de la memoria
                        ;de programa.
        setf  0x20,0    ;Esta instrucción no se ejecuta ya que se encuentra en la
                        ;dirección 0x26 de la flash.
FIN:
        goto FIN        ;Esta instrucción se ejecuta después del bc, ya que corresponde a
                        ;la dirección 0x28.
```

2. **Instrucción: bn n**

Descripción:

Esta instrucción salta a la dirección de la memoria de programa especificada en n (el PC toma el valor n) si es que el bit Negativo (N) del registro STATUS es '1'. En caso contrario, no se produce ningún salto.

Operandos:

n -> Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si se produce el salto. 1 ciclo de instrucción si el bit *Carry* es '0'.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .5        ;Se carga el registro W con el valor decimal .5
        movwf 0x40,0    ;Se guarda el valor .5 en el registro especificado por la
                        ;dirección 0x40 de la SRAM
        negf  0x40,0    ;Se obtiene el Complemento a 2 de .5, resultando -5 (11111101₂)
        bn    LAZO      ;Dado que el resultado es negativo, el bit N se pone a '1' y se
                        ;produce el salto a LAZO.
        clrf  0x40,0    ;Esta instrucción no se ejecuta.
LAZO:
        goto LAZO       ;El PC toma el valor de LAZO y se ejecuta esta instrucción
                        ;ininterrumpidamente.
```

3. **Instrucción: bnc n**

Descripción:

Esta instrucción salta a la dirección de la memoria de programa especificada en n (el PC toma el valor n) si es que el bit *Carry* (C) del registro STATUS es '0'. En caso contrario, no se produce ningún salto.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si se produce el salto. 1 ciclo de instrucción si el bit *Carry* es '1'.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .255          ;Se carga el valor decimal .255 en el registro W
        movwf 0x40,0        ;Se coloca el valor .255 en el registro que ocupa la dirección
                            ;0x40 de la SRAM
        incf  0x40,0        ;Se incrementa el registro en 1, produciéndose un
                            ;desbordamiento (Carry = '1')
        bnc   LAZO          ;Si el Carry = '0' se salta a la etiqueta LAZO. Como no es el
                            ;caso, no hay salto
        clrf  0x40,0        ;Se ejecuta la instrucción y el registro de la dirección 0x40 de la
                            ;SRAM se pone a 0.
LAZO:
        goto LAZO           ;Finalmente se ejecuta esta instrucción, pero, sin haber realizado
                            ;un salto previo
```

4. **Instrucción: bnn n**

Descripción:

Esta instrucción salta a la dirección de la memoria de programa especificada en n (el PC toma el valor n) si es que el bit Negativo (N) del registro STATUS es '0'. En caso contrario, no se produce ningún salto.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si se produce el salto. 1 ciclo de instrucción si el bit *Carry* es '1' lógico.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .5            ;Se carga el valor decimal .5 en el registro W.
        movwf 0x40,0        ;El valor .5 se guarda en la dirección 0x40 de la SRAM.
        negf  0x40,0        ;Se niega el valor guardado en memoria, obteniéndose -5
                            ;(Complemento a 2).
        bnn   LAZO          ;Dado que el bit N = '1', no hay salto y por tanto se ejecuta la
                            ;siguiente instrucción.
        clrf  0x40,0        ;Se pone a 0 el registro apuntado por la dirección 0x40 de la
                            ;SRAM.
LAZO:
        goto LAZO           ;Se ejecuta el salto indefinido a la etiqueta LAZO.
```

5. **Instrucción: bnov n**

Descripción:

Esta instrucción salta a la dirección de la memoria de programa especificada por n (el PC toma el valor n) si es que el bit *Overflow* (OV) del registro STATUS es '0'. En caso contrario, no se produce ningún salto.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si no ha ocurrido un evento de *Overflow*. 1 ciclo de instrucción si ha ocurrido previamente un *Overflow*.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .100          ;Se carga el valor decimal .100 en el registro W
        movwf 0x20,0        ;Se carga el valor .100 de W al registro de la dirección 0x20 de la
                            ;SRAM
        movlw .50           ;Se carga el valor decimal .50 en el registro W
        addwf 0x20,f,0      ;Se suma el valor .100 de la dirección SRAM con .50 del
                            ;registro W. El resultado es .150, el cual tiene el bit MSB = '1'.
        bnov  LAZO          ; Por lo tanto OV = '1' y no hay salto.
        clrf  0x20,0        ;Se pone a 0 el registro de la dirección 0x20 de la memoria
                            ;SRAM
LAZO:
        goto LAZO           ;Se ejecuta el salto a la etiqueta LAZO de forma indefinida.
```

6. **Instrucción: bnz n**

Descripción: Esta instrucción evalúa si el bit Cero (Z) del registro STATUS es '0' lógico. De ser afirmativa esta condición, se produce un salto a la dirección de la memoria de programa especificada por n. En caso contrario, si el bit Z es '1' lógico, no habrá salto y se ejecutará la siguiente instrucción.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si no ha ocurrido un evento de resultado cero (Z='0'). 1 ciclo de instrucción si ha ocurrido previamente un resultado cero (Z='1').

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .100      ;Se carga el registro W con el valor decimal .100
        sublw .100      ;Se resta el valor del registro W del factor decimal .100,
                        ;obteniéndose 0 de resultado
        bnz   LAZO      ;El bit Z = '1' y por tanto no hay salto a la etiqueta LAZO
        movlw .10       ;Se ejecuta esta instrucción y el registro W se carga con el valor
                        ;decimal .10.
LAZO:
        goto LAZO       ;Luego se ejecuta esta instrucción y se mantiene el salto a LAZO
                        ;de forma indefinida.
```

7. **Instrucción: bov n**

Descripción:

Esta instrucción salta a la dirección de la memoria de programa especificada por n (el PC toma el valor n) si es que el bit *Overflow* (OV) del registro STATUS es '1'. En caso contrario, no se produce ningún salto.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si ha ocurrido un evento de *Overflow*. 1 ciclo de instrucción si no ha ocurrido previamente un *Overflow*.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .100          ;Se carga el valor decimal .100 en el registro W
        movwf 0x20,0        ;Se carga el contenido de W al registro de la dirección 0x20 de la
                            ;SRAM.
        movlw .50           ;Se carga el valor decimal .50 al registro W.
        addwf 0x20,f,0      ;Se suma el valor .100 con .50 y se obtiene como
                            ;resultado .150 en la memoria SRAM.
        bov  LAZO           ;Como ha ocurrido un Overflow, OV = '1' y por tanto se salta a
                            ;la etiqueta LAZO.
        clrf  0x20,0        ;Esta instrucción no se ejecuta.
LAZO:
        goto LAZO           ;Se ejecuta esta instrucción de salto de manera indefinida.
```

8. **Instrucción: bra n**

Descripción:

Esta instrucción realiza un salto a la dirección especificada por n. El salto es de corto alcance, lo que significa que puede ser de ±512 instrucciones desde la posición donde se ejecuta la instrucción bra.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .100          ;Se carga el registro W con el valor decimal .100
        movwf 0x20,0        ;Se envía el valor decimal .100 al registro de la dirección 0x20 de
                            ;la SRAM
        bra   LAZO          ;Se salta a la dirección especificada por la etiqueta LAZO (solo
                            ;se salta 2 instrucciones)
        clrf  0x20,0        ;No se ejecuta esta instrucción
        incf  0x20,0        ;No se ejecuta esta instrucción
LAZO:
        goto LAZO           ;El salto lleva a la ejecución directa de la instrucción goto de
                            ;manera ininterrumpida.
```

9. **Instrucción: bz n**

Descripción:

Esta instrucción permite realizar un salto a la dirección de la memoria de programa especificada por n si es que el bit Cero (Z) del registro STATUS es '1'.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

Duración: 2 ciclos de instrucción si el bit Z = '1'. 1 ciclo de instrucción si el bit Z = '0'.

Bits del registro STATUS afectados: Ninguno

Ejemplo:

```
        movlw .100      ;Se carga el registro W con el valor decimal .100
        sublw .100      ;Se restan dos valores iguales y el resultado da 0. Por tanto el bit
                        ;Z = '1'
        bz    LAZO      ;Se realiza el salto a la etiqueta LAZO dado que el bit Z = '1'
        movlw .10       ;Por lo tanto, esta instrucción no se ejecuta
LAZO:
        goto LAZO       ;Se salta a esta instrucción y se ejecuta de manera indefinida
```

10. **Instrucción: call n, s**

Descripción:

Esta instrucción realiza un salto a la dirección de la memoria de programa especificada en n. La dirección del PC + 2 de la instrucción *call* se guarda en la pila, con lo cual al detectar una instrucción *Return* en un futuro, se podrá retornar a la siguiente instrucción después del *call* para su ejecución.

Operandos:

n ->Dirección de la memoria de programa a la cual se hará el salto (PC = n). n también puede ser el nombre de una etiqueta.

s -> Si es 0, los valores de los registros BSR, W y STATUS no se guardan en la pila de llamado rápido. Si es 1, entonces se guardan automáticamente los valores actuales de BSR, W y STATUS, los cuales pueden ser restablecidos al momento de ejecutar la instrucción *Return*.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo 1:

```
        movlw .10           ;Se carga el valor de decimal .10 en el registro W.
        movwf 0x40,0        ;Se guarda el valor .10 en la dirección 0x40 de la SRAM.
        call  SUMA          ;Se salta a la dirección especificada por la etiqueta SUMA.
        bc    FIN           ;Se salta a la etiqueta FIN, ya que el bit C='1'.
        setf  0x40,0        ;Esta instrucción no se ejecuta.
FIN:
        goto FIN            ;Se ejecuta esta instrucción de forma indefinida.
SUMA:
        movlw .246          ;Se carga en el registro W el valor decimal .246.
        addwf 0x40,f,0      ;Se suma .246 con el valor de .10 de la dirección 0x40. El
                            ;resultado es 0 y C = '1'.
        return              ;Se retorna del último call con el bit C = '1'.
```

Ejemplo 2:

```
        movlw .10           ;Se carga en el registro W el valor decimal .10. STATUS = 0,
                            ;W= .10
        movwf 0x40,0        ;Se guarda el valor .10 en la dirección 0x40 de la SRAM
        call  SUMA,1        ;Se salta a la etiqueta SUMA, pero guardando en la pila W, BSR
                            ;y STATUS
        bc    FIN           ;Dado que el registro STATUS antes del call era 0, entonces C =
                            ;'0' y no hay salto.
        movwf  0x40,0       ;Se carga el valor del registro W restablecido (.10) en la
                            ;dirección 0x40 de la SRAM
FIN:
        goto FIN            ;Finalmente, se ejecuta esta instrucción de forma indefinida.
SUMA:
        movlw .246          ;Se carga en el registro W el valor decimal .246
        addwf 0x40,f,0      ;Se suma el valor .246 con el valor de .10, generando como
                            ;resultado 0 y C = '1'.
        return 1            ;Se retorna a la instrucción después del call, pero,
                            ;restableciendo STATUS, W y BSR.
```

11. **Instrucción: clrwdt**

Descripción:

Esta instrucción inicializa el contador del *Watch Dog Timer*, evitando así que el CPU entre en Reset.

Operandos: No tiene.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
clrwdt   ; Se inicializa el contador del Watch dog Timer.
```

12. **Instrucción: daw**

Descripción:

Esta instrucción realiza el ajuste decimal de una cifra para que pueda convertirse al formato BCD. El ajuste decimal consiste en sumar 6 a todo *nibble* cuyo código hexadecimal sea mayor que 0xA o genere un acarreo en la siguiente cifra más significativa. Por ejemplo, si se suman los valores 0x78 + 0x14 en BCD, la respuesta es 0x92. Sin embargo, este resultado, al ejecutarse de manera binaria, daría 0x8C. La instrucción DAW transforma el 0x8C en 0x92, dándole su respectivo formato BCD.

Operandos: No tiene.

Duración: 1 ciclos de instrucción.

Bits del registro STATUS afectados: C

Ejemplo:

```
        movlw 0x58          ;Se carga el valor hexadecimal 0x58 en el registro W
        movwf 0x40,0        ;Se carga el valor 0x58 en la dirección 0x40 de la SRAM
        movlw 0x15          ;Se carga el valor hexadecimal 0x15 en el registro W
        addwf 0x40,W,0      ;Se realiza la suma. El resultado entrega el valor
                            ;0x58+0x15 = 0x6D
        daw                 ;Con el ajuste decimal W = 0x73, la suma en formato BCD.
FIN:
        goto FIN            ;Se ejecuta la instrucción de salto de manera indefinida.
```

13. **Instrucción: goto n**

Descripción:

Esta instrucción realiza un salto a la dirección de la memoria de programa especificada por n. n puede ser cualquier valor de la memoria de programa. Aquí no existe retorno, no hay apilamiento del PC y simplemente este registro adquiere el valor de n.

Operandos:

n -> dirección a la cual se va a realizar el salto.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
        org 0x00        ;se define el vector de Reset y se ubica la instrucción goto MAIN
                        ;en esta dirección
        goto MAIN       ;al iniciar el programa se ejecuta esta instrucción, saltando a la
                        ;posición 0x100 de la memoria de programa.
        org 0x100
MAIN:
        clrf 0x40,0     ;En la dirección 0x100 se encuentra esta instrucción la cual es
                        ;ejecutada.
```

14. **Instrucción: nop**

Descripción:

Instrucción de no operación. Simplemente no se ejecuta ningún proceso del microprocesador, pero se pierde un ciclo de instrucción en esta operación inocua.

Operandos: No hay operandos.

Duración: 1 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
movlw .10       ;Se carga el valor decimal .10 en el registro W
nop             ;Se deja pasar un ciclo de instrucción y no se ejecuta ninguna
                ;acción.
nop             ;Se deja pasar un ciclo de instrucción y no se ejecuta ninguna
                ;acción.
addlw .20       ;Se incrementa en .20 el valor de W después de 3 ciclos de
                ;instrucción.
```

15. **Instrucción: push**

Descripción:

Esta operación permite almacenar en la pila la dirección de la flash siguiente (PC+2) de la instrucción push. Luego, se aumenta en un dígito el puntero de pila (STKPTR), de tal manera que se apunta a una dirección más alta.

Operandos: No tiene.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
        org 0x00            ;Se fija el vector de Reset con la instrucción goto MAIN
        goto MAIN           ;La instrucción goto MAIN lleva el PC a la dirección 0x20
        org 0x20            ;Se establece la dirección 0x20 de la memoria de programa
MAIN:
        movlw 0x58          ;Se almacena esta instrucción en la dirección 0x20
        movwf 0x40,0        ;Se almacena esta instrucción en la dirección 0x22
        push                ;Esta instrucción se almacena en la dirección 0x26 y este último
                            ;valor se guarda en la pila
        movlw 0x15          ;Se almacena esta instrucción en la dirección 0x26
```

16. **Instrucción: pop**

Descripción:

Esta operación permite disminuir en un dígito el puntero de pila (STKPTR), de tal manera que se apunta a una dirección más baja de la pila.

Operandos: No cuenta con operandos.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
        org 0x00            ;Se fija el vector de Reset con la instrucción goto MAIN
        goto MAIN           ;La instrucción goto MAIN lleva el PC a la dirección 0x20
        org 0x20            ;Se establece la dirección 0x20 de la memoria de programa
MAIN:
        movlw 0x58          ;Se almacena esta instrucción en la dirección 0x20
        movwf 0x40,0        ;Se almacena esta instrucción en la dirección 0x22
        push                ;Esta instrucción se almacena en la dirección 0x26 y este último
                            ;valor se guarda en la pila
        movlw 0x15          ;Se almacena esta instrucción en la dirección 0x26
        pop                 ;Disminuye el puntero de pila haciendo que apunte a la dirección
FIN:                        ;inicial.
        goto FIN            ;Se ejecuta esta instrucción de forma indefinida.
```

17. **Instrucción: rcall n**

Descripción: Instrucción de salto relativo a la dirección n. El valor de n no puede alejarse más de ±512 instrucciones de la posición de la instrucción *rcall*.

Operandos:

n ->Dirección de la memoria de programa que tomará el PC para el salto a esa posición.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
        movlw .5            ;Se carga el valor decimal .5 en el registro W
        addlw .15           ;Se suma el valor .15 con .5 obteniéndose el valor .20
        movwf 0x40,0        ;Se guarda el valor .20 en la dirección 0x40 de la SRAM
        rcall ANALIZA       ;Se salta a la instrucción ubicada en la dirección de la
                            ;etiqueta ANALIZA
        movf  0x40,W,0      ;Se carga en W el valor de la dirección 0x40, el cual es 0.
FIN:
        goto FIN            ;Se ejecuta la instrucción goto de manera indefinida.
ANALIZA:
        btfss 0x40,0,0      ;Se verifica si el bit LSB del registro de la dirección 0x40 es 0.
                            ;Condición falsa.
        clrf  0x40,0        ;No se realiza el salto y se pone a 0 la dirección 0x40.
        return              ;Se retorna a la instrucción siguiente al rcall.
```

18. **Instrucción: reset**

Descripción:

Esta instrucción realiza un Reset por *software* del CPU. Luego de esta instrucción, el programa se ejecuta desde la dirección 0x00 de la memoria FLASH.

Operandos: No tiene.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
movlw .6            ;Se carga el valor decimal .6 en el registro W
addlw .4            ;Se suma el valor decimal .4 al .6 y se obtiene el valor .10
                    ;en el registro W
reset               ;Se aplica una instrucción de Reset y el programa se reinicia
movlw .5            ;Esta instrucción nunca se ejecuta dado que la operación de Reset
                    ;se ejecuta antes
```

19. **Instrucción: retfie s**

Descripción:

Esta instrucción permite el retorno de una rutina de interrupción al flujo normal del programa. El valor s permite restablecer los registros BSR, W y STATUS antes de la interrupción.

Operandos:

s -> operador que indica si el retorno recuperará los valores de W, BSR y STATUS que se guardaron en el evento de interrupción cuando es 1. Si este operador es 0, los valores de estos registros no se guardan de forma automática.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
RUTINA_INT:
        movlw 0x90      ;Se carga el registro W con el valor 0x90
        movwf LATD      ;Se muestra el valor 0x90 en el Puerto D
        retfie          ;Se retorna de la interrupción.
```

20. **Instrucción: return s**

Descripción:

Esta instrucción permite el retorno de una operación *call* o *rcall*. Devuelve al PC su valor anterior PC+2 cuando se realizó la rutina *call* o *rcall*.

Operandos:

s -> operador que indica si el retorno recuperará los valores de W, BSR y STATUS que se guardaron en el evento de interrupción cuando es 1. Si este operador es 0, los valores de estos registros no se guardan de forma automática.

Duración: 2 ciclos de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
MAIN:
        call RUTINA     ;Se salta a la dirección especificada por la etiqueta RUTINA.
LAZO:
        goto LAZO       ;Se repite esta instrucción de salto de forma indefinida.
RUTINA:
        movlw .10       ;Se carga el valor .10 en el registro W.
        movwf 0x40,0    ;Se carga el valor .10 en la dirección 0x40.
        return          ;Se retorna del call, ejecutando la instrución goto LAZO.
```

21. **Instrucción: sleep**

Descripción:

Esta instrucción pone al microcontrolador en modo de bajo consumo. Algunos periféricos y funciones quedan deshabilitadas y solo puede despertar bajo ciertos eventos de interrupción.

Operandos: No tiene.

Duración: 1 ciclo de instrucción.

Bits del registro STATUS afectados: Ninguno.

Ejemplo:

```
sleep    ; Se activa el modo de bajo consumo del microcontrolador.
```

4.2 Modos de direccionamiento

Como se puede notar del análisis de las instrucciones del PIC18F, existen diversas operaciones relacionadas con la manipulación de datos en la memoria de programa y memoria SRAM. La forma como se accede mediante una instrucción a una dirección de memoria se conoce como un Modo de direccionamiento.

Los modos de direccionamiento requieren el conocimiento de la estructura de la memoria SRAM de datos. Si se observa el gráfico 2.8, allí se mostró un esquema de la memoria SRAM. Como se pudo ver, en el espacio de memoria específico del PIC18F4550 solamente se cuenta con 8 bancos de 256 bytes cada uno. La zona de los bancos 8 al 14 no se encuentra en uso y el banco 15 contiene los registros de función especial, los cuales se acceden directamente mediante su dirección absoluta. Muchas de las instrucciones de manejo de memoria no utilizan operandos de 12 bits, por lo cual no es posible especificar una dirección absoluta en una sola instrucción. Para esto, el registro BSR permite seleccionar cada uno de los bancos y con una dirección de 8 bits es posible direccionar los 256 registros de cada banco.

Para el PIC18F existen cuatro modos de direccionamiento que permiten la manipulación de todos los registros del microcontrolador: inherente, directo, indirecto y literal.

4.2.1 Modo de direccionamiento inherente

En el modo de direccionamiento inherente la instrucción no presenta operando, el mnemónico por sí solo especifica la dirección de memoria afectada. Son solo tres instrucciones del PIC18F que cumplen con este modo de direccionamiento:

```
1. daw      ;(Ajuste decimal del valor contenido en el registro W, el
            ;resultado se almacena en el mismo registro, por lo tanto, no es
            ;necesario el operando)
2. reset    ;(Reset por software del CPU)
3. nop      ;(Sin operación, solo se genera un retardo de un ciclo de
            ;instrucción)
```

4.2.2 Modo de direccionamiento literal

En el modo de direccionamiento literal el valor del operando es indicado explícitamente en la instrucción. Aquí se tienen más instrucciones que en el caso anterior; de hecho, todas las instrucciones de tipo literal manejan este modo de direccionamiento. Además, en las instrucciones de salto el operando indica la dirección que será cargada en el registro PC. Algunas instrucciones con modo de direccionamiento literal son las siguientes:

```
1. addlw 0x45        ;(Se suma el valor guardado en el registro W con el valor literal
                     ;0x45)
2. goto 0x100        ;(Se realiza el salto a la dirección de memoria de programa
                     ;0x100)
3. movlw .15         ;(Se carga en el registro W el valor decimal .15)
```

4.2.3 Modo de direccionamiento directo

En el modo de direccionamiento directo la dirección de la memoria de datos donde se encuentra el operando viene explícitamente indicada en la instrucción. El operando puede ser un byte o un bit de una dirección de memoria.

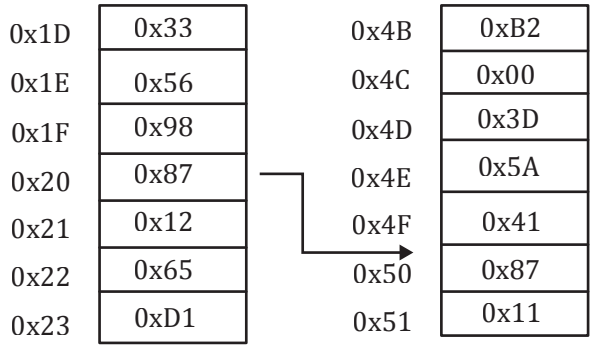
La instrucción movff fs, fd, donde fs y fd son la dirección origen y destino. Por ejemplo, la instrucción movff 0x20, 0x50 carga el contenido de la dirección 0x20 en la dirección 0x50. Tal como se muestra en el esquema del gráfico 4.2. Ambos operandos representan direcciones.

Por otro lado, también es posible hacer uso del modo de direccionamiento directo mediante el registro selector de banco (BSR) y la dirección relativa. En el ejemplo mostrado en el programa 4.3 se carga el valor decimal .1 en el BSR (seleccionando el banco 1 que inicia en la dirección absoluta 0x100), luego se carga el valor binario 01011100_2 al registro W, y este es ingresado a la dirección relativa al banco 1 0x20 (dirección absoluta 0x120). En el gráfico 4.3 se muestra el esquema de la manipulación de la memoria de datos. Como se sabe, la instrucción movwf solo soporta un operando de 8 bits, por lo cual, la dirección debe ser un valor entre 0 y 255.

Programa 4.3. Modo de direccionamiento directo en base a la dirección relativa

```
movlb   .1              ;Se selecciona el banco 1 con BSR = .1.
movlw b'01011100'       ;Se carga en el registro W el valor binario 01011100₂
movwf 0x20,1            ;Se guarda en la dirección 0x20 relativa al banco 1 el valor
                        ;0101 1100 que corresponde a la dirección absoluta 0x100 +
                    ;   0x20 =0x120
```

Gráfico 4.2. Esquema de funcionamiento de la instrucción movff 0x20,0x50



Elaboración propia.

El direccionamiento directo también se realiza en el modo absoluto. Aquí se asume por defecto que todas las direcciones se encuentran en el banco 0 y no es necesario especificar el valor del registro BSR. Solo es posible acceder a las 256 primeras direcciones de la memoria de datos. Por ejemplo, en el programa 4.4 se muestra una secuencia simple de direccionamiento directo absoluto. En el gráfico 4.4 se ofrece el esquema de manipulación de memoria.

Programa 4.4. Modo de direccionamiento directo en base a la dirección absoluta

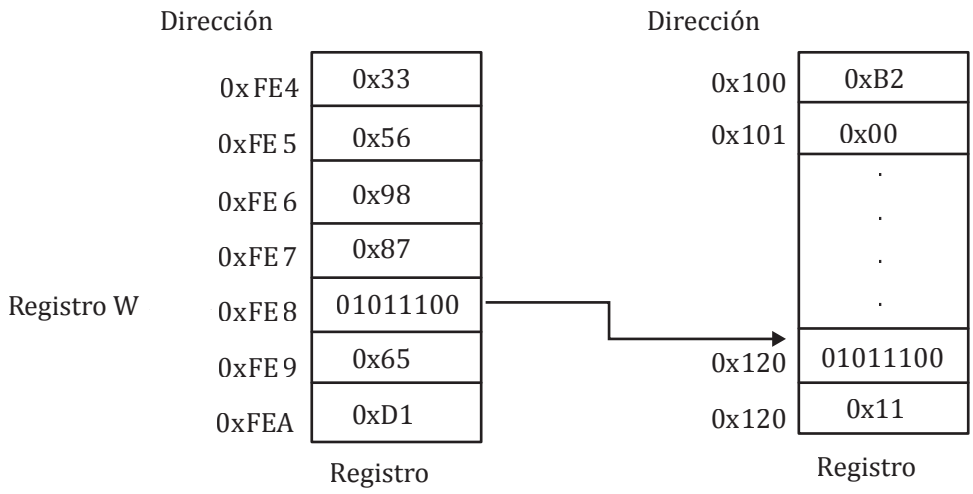
```
movlw   .45          ;Se carga en el registro W el valor decimal .45
movwf  0x60,0        ;El valor se ubica en la dirección 0x60 absoluta de la memoria de
                     ;datos.
```

Finalmente, el acceso a los bits de una dirección se logra con las instrucciones orientadas a bit que también corresponden a un modo de direccionamiento directo. En este tipo de operación se debe especificar la dirección de la memoria de datos, el orden del bit (de 0 a 7) y si el acceso es relativo o absoluto. En el programa 4.5 se muestra un ejemplo de este tipo de direccionamiento, y en el gráfico 4.5 se ofrece su esquema de manipulación de memoria de datos.

Programa 4.5. Modo de direccionamiento directo para modificación de bit de una dirección de memoria de datos

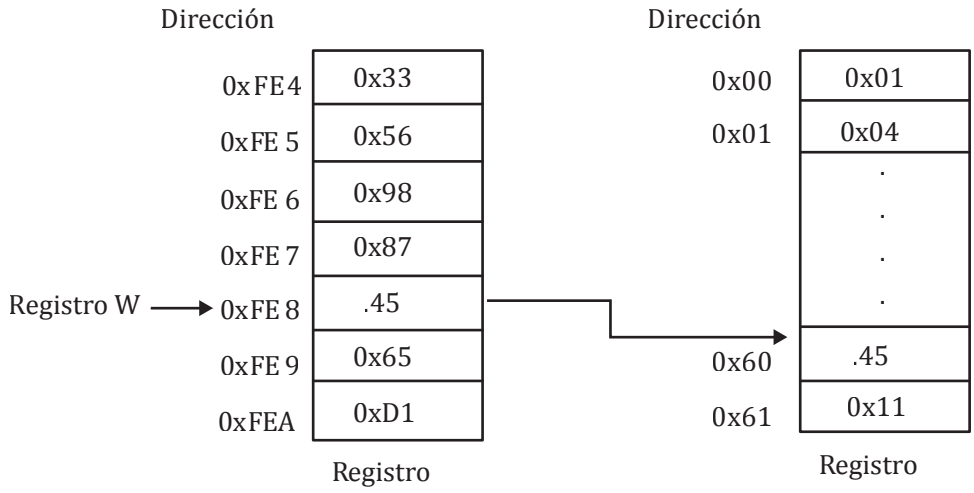
```
movlb    .2          ;Se selecciona el Banco 2 de la memoria de datos
bsf     0x30,7,1     ;El bit 7 de la dirección 0x30 + 0x200 = 0x230 se pone a nivel '1'
```

Gráfico 4.3. Esquema de manipulación de memoria haciendo uso del direccionamiento directo relativo al banco



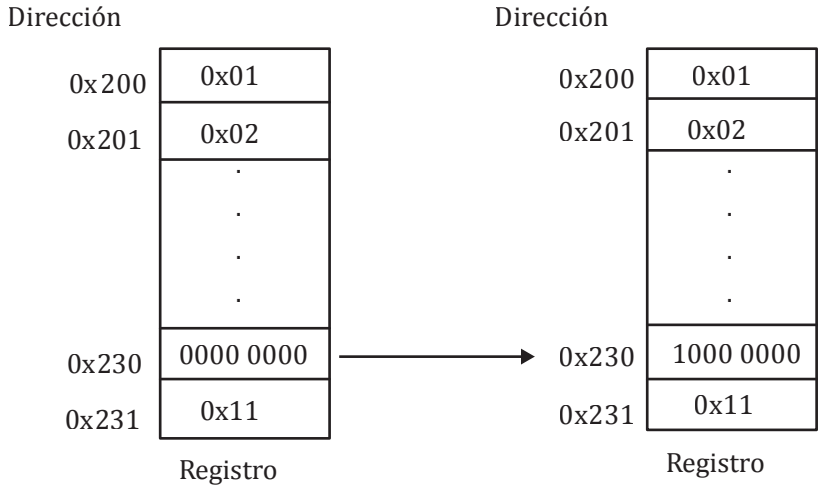
Elaboración propia.

Gráfico 4.4. Esquema de manipulación de memoria haciendo uso del direccionamiento directo absoluto



Elaboración propia.

Gráfico 4.5. Esquema de manipulación de bits de memoria haciendo uso del direccionamiento directo



Elaboración propia.

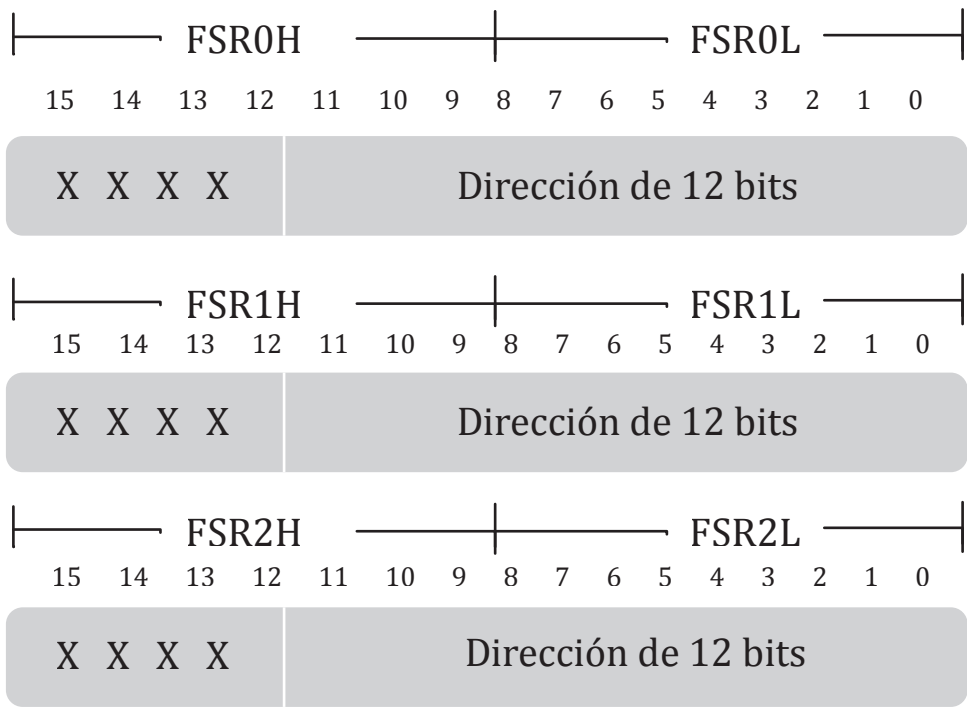
4.2.4 Modo de direccionamiento indirecto

En el modo de direccionamiento indirecto se hace uso de los registros FSRx e INDFx (donde x puede ser 0, 1 o 2). Los registros FSRx están compuestos por 12 bits y se encargan de apuntar a la dirección de la memoria de datos a la cual se quiere acceder para un proceso de lectura o escritura. Estos registros se componen de dos partes: FSRxH y FSRxL. El primer registro contiene los 4 bits más significativos de la dirección, mientras que el segundo registro contiene los 8 bits menos significativos. Estos registros forman parte de los RFE del mapa de memoria. En el gráfico 4.6 se puede observar un esquema de estos registros.

Los registros INDFx apuntan al contenido direccionado por los registros FSRx. Por ejemplo, si el registro FSR0 toma el valor de 0x203, y se realiza una escritura al registro INDF0, entonces se estará cargando este valor escrito en la dirección 0x203. Lo mismo aplica para un acceso de lectura. Con la combinación de accesos al registro FSRx e INDFx se obtiene el modo de direccionamiento indirecto, ya que indirectamente se están apuntando a las direcciones de memoria y a su contenido.

Si se analiza el programa 4.6, se podrá observar que se inicializa la dirección apuntada por el registro FSR0 con el valor 0x200. La dirección absoluta 0x20 actúa como variable, almacenando inicialmente el valor decimal .10. Luego, este último valor es almacenado en la dirección 0x200, ya que se carga el contenido de la dirección 0x20 en el registro INDF0 (el cual representa el contenido de la dirección 0x200). Posteriormente, se incrementa la parte baja del registro FSR0 (registro FSR0L), logrando que ahora se apunte a la dirección 0x201. Se resta el valor de la dirección absoluta 0x20, quedando en .9. Ya que este valor es diferente de 0, el proceso se repite nuevamente, una y otra vez hasta que la dirección absoluta 0x20 contenga el valor de 0. En el gráfico 4.7 se observa un esquema de los datos ingresados en memoria por el programa 4.6.

Gráfico 4.6. Registros FSRx para el modo de direccionamiento indirecto



Elaboración propia.

Programa 4.6. Ejemplo básico del modo de direccionamiento indirecto

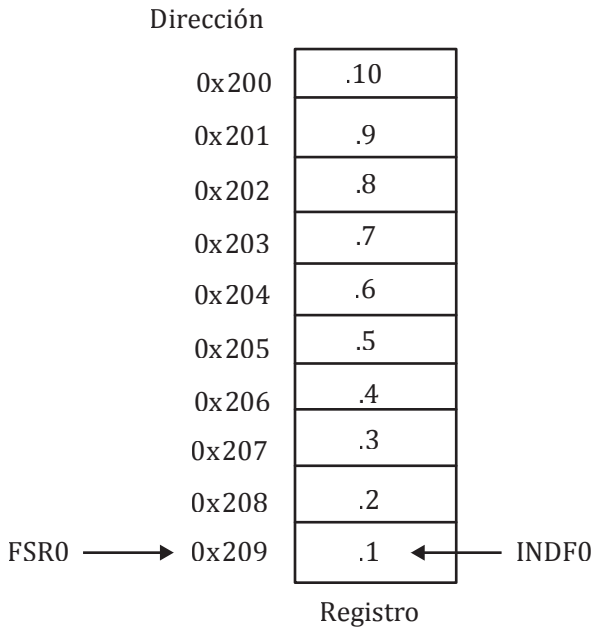
```
movlw .10          ;Se carga el valor decimal .10 en el registro W
movwf 0x20,0       ;Se copia el contenido del registro W a la dirección absoluta
                   ;0x20
lfsr FSR0,0x200    ;Se carga el registro FSR0 con la dirección 0x200
```

```
LAZO:
        movf 0x20,W,0       ;Se carga el contenido de la dirección absoluta 0x20 en el
                            ;registro W
        movwf INDF0         ;El contenido de W ingresa al registro INDF0, por tanto se
                            ;guarda en la dirección 0x200
        incf  FSR0L         ;Se incrementa la dirección apuntada por FSR0L en un dígito
        decfsz  0x20,f,0    ;Se resta el contenido de la dirección absoluta 0x20 y se
                            ;compara con 0
        goto  LAZO          ;Si es diferente de 0 se salta a la etiqueta LAZO
BUCLE:
        goto BUCLE          ;Si el contenido es 0 se ingresa a este bucle indefinido
```

Existen también otros registros de función especial que permiten simplificar la manipulación de memoria de datos en el modo de direccionamiento indirecto. Por ejemplo, los registros POSTEDC0, POSTEDC1 y POSTEDC2 posibilitan leer o grabar en el contenido del registro INDF y luego decrementar en 1 el valor del registro FSR. Esto evita tener que realizar dos instrucciones: una para la lectura o escritura del INDF y otra para decrementar el registro FSR.

En el programa 4.7 se agrega más código al programa 4.6. Una vez llenada la memoria de datos desde las direcciones 0x200 a la 0x209 (tal como se muestra en el gráfico 4.7), se accede al registro POSTDEC0 para la lectura de los bytes de estas direcciones y se vuelven a almacenar en el rango de direcciones 0x280 a 0x289. En el gráfico 4.8 se muestra el esquema de las dos regiones de memoria direccionadas indirectamente.

Gráfico 4.7. Esquema de la memoria de datos luego del acceso mediante direccionamiento indirecto en el programa 4.6

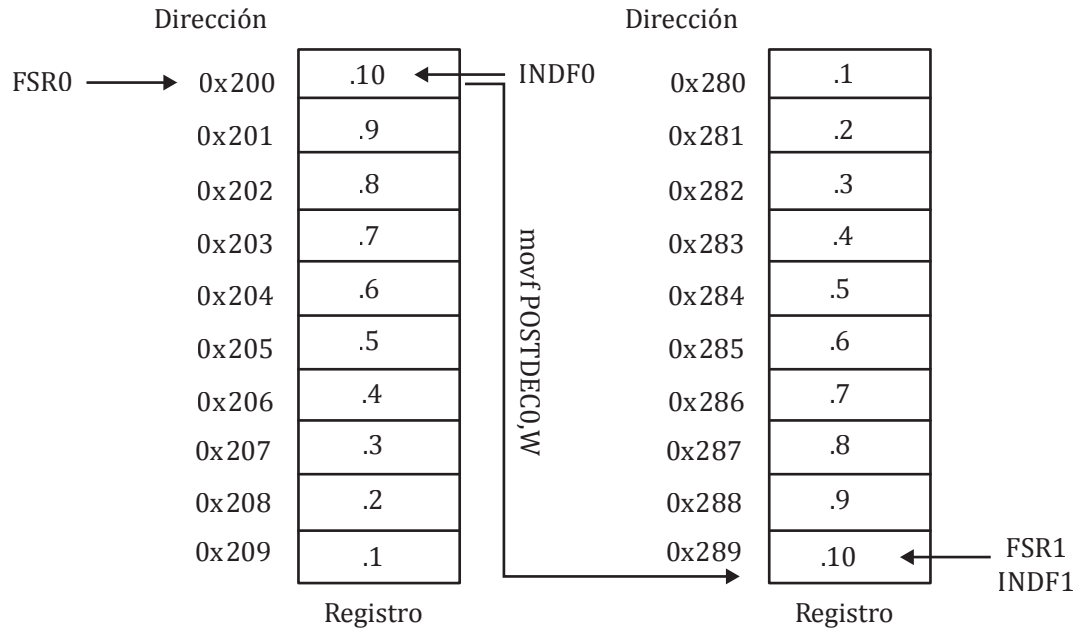


Elaboración propia.

Programa 4.7. Uso del registro POSTDECx para la escritura de un rango de direcciones de memoria

```
        movlw .10           ;Se carga el registro W con el valor decimal .10
        movwf 0x20,0        ;Se guarda el valor .10 en la dirección 0x20 de la memoria
                            ;de datos
        lfsr FSR0,0x200     ;Se carga el registro FSR0 con la dirección absoluta 0x200
LAZO:
        movf 0x20,W,0       ;Se lee el contenido de la dirección absoluta 0x20
        movwf INDF0         ;Se carga el contenido almacenado en la dirección 0x20 en
                            ;la dirección FSR0
        incf  FSR0L         ;Se incrementa FSR0L para apuntar a una dirección
                            ;superior
        decfsz  0x20,f,0    ;Se resta en 1 el contenido de la dirección 0x20 y se
                            ;compara con 0
        goto  LAZO          ;Si no es 0, entonces se repite el proceso
        lfsr FSR1,0x280     ;De ser 0, se carga en el registro FSR1 la dirección absoluta
                            ;0x280
        decf  FSR0L         ;Se resta en .1 FSR0 para que apunte a la dirección 0x209.
BUCLE:
        movf POSTDEC0,W     ;Se carga en W el valor INDF0 para FSR0 = 0x209.
                            ;Luego FSR0 = FSR0-1 = 0x208.
        movwf INDF1         ;Se guarda el valor leído de la dirección FSR0 en la
                            ;dirección apuntada por FSR1.
        movwf 0x20,0        ;Se carga el valor leído de la dirección FSR0 en la
                            ;dirección absoluta 0x20.
        incf  FSR1L          ;Se incrementa el puntero de direcciones FSR1 en 1.
        movlw  .10          ;Se carga el valor .10 en el registro W.
        cpfseq 0x20,0       ;Se compara el valor almacenado en 0x20 con .10 (El
                            ;último valor a leer y copiar).
        goto   BUCLE        ;Si no es .10, entonces se repite el proceso.
FIN:
        goto FIN            ;De ser .10 el valor leído se terminó de copiar el bloque de
                            ;memoria y se finaliza con un bucle indefinido.
```

Gráfico 4.8. Esquema de la memoria de datos luego del acceso mediante direccionamiento indirecto y haciendo uso del registro POSTDEC0 en el programa 4.7



Elaboración propia.

Los registros POSTINCx (donde x puede ser 0, 1 o 2) realizan una función similar a los registros POSTDECx, con la diferencia de que en este caso se accede al contenido de la dirección de memoria (ya sea para lectura o escritura) y luego el registro FSRx se autoincrementa. En el programa 4.8 se muestra un ejemplo equivalente al programa 4.7, pero haciendo uso del registro POSTINC1 al momento de escribir los datos en el rango de direcciones 0x280 a 0x289.

Programa 4.8. Uso del registro POSTINCx para la escritura de un rango de direcciones de memoria

```
        movlw .10           ;Se carga el registro W con el valor decimal .10.
        movwf 0x20,0        ;Se guarda el valor .10 en la dirección 0x20 de la memoria
                            ;de datos.
        lfsr FSR0,0x200     ;Se carga el registro FSR0 con la dirección absoluta 0x200.
LAZO:
        movf 0x20,W,0       ;Se lee el contenido de la dirección absoluta 0x20.
        movwf INDF0         ;Se carga el contenido almacenado en la dirección 0x20 en
                            ;la dirección FSR0.
        incf  FSR0L         ;Se incrementa FSR0L para apuntar a una dirección
                            ;superior.
        decfsz  var,f,0     ;Se resta en 1 el contenido de la dirección 0x20 y se
                            ;compara con 0.
        goto  LAZO          ;Si no es 0, entonces se repite el proceso:
```

```
        lfsr FSR1,0x280         ;De ser 0, se carga en el registro FSR1 la dirección absoluta
                                ;0x280.
        decf  FSR0L             ;Se resta en 1 FSR0 para que apunte a la dirección 0x209.
BUCLE:
        movf POSTDEC0,W         ;Se carga en W el valor INDF0 para FSR0 = 0x209.
                                ;Luego FSR0 = FSR0-1 = 0x208.
        movwf POSTINC1          ;Se carga el valor de W en el registro INDF1 y se
                                ;incrementa FSR1.
        movwf 0x20,0            ;Se carga el valor leído de la dirección FSR0 en la dirección
                                ;absoluta 0x20
        movlw  .10              ;Se carga el valor .10 en el registro W
        cpfseq 0x20,0           ;Se compara el valor almacenado en 0x20 con .10 (El
                                ;último valor a leer y copiar).
        goto   BUCLE            ;Si no es .10, entonces se repite el proceso.
FIN:
        goto FIN                ;De ser .10 el valor leído se terminó de copiar el bloque de
                                ;memoria y se finaliza con un bucle indefinido.
```

Existen también otros registros, como los PREINCx y PLUSWx, que realizan funciones similares a POSTDECx y POSTINCx, respectivamente, con la diferencia de que el acceso a PREINCx primero decrementa el registro FSRx y luego accede a INDFx, mientras que el acceso al registro PLUSWx primero incrementa el registro FSRx y luego accede a INDFx.

4.3 El Puntero de programa

El Puntero de programa o Contador de programa (PC) es un registro de 21 bits que siempre apunta a la dirección de la memoria de programa donde se encuentra la instrucción que será ejecutada. Dado que el PC siempre presenta su bit LSB en '0', este apuntará a direcciones pares. El PC se encuentra formado por tres registros PCU, PCH y PCL. El primero contiene los 5 bits más significativos (bits 20 al 16), el segundo los bits 15 al 8 y el último los bits 7 al 0, respectivamente.

El PC no se puede acceder de forma directa mediante las instrucciones del PIC18F. Como se vio en el capítulo 2, existen tres registros de función especial de 8 bits: PCL (que apunta a los 8 bits menos significativos del PC), PCLATH (que apunta a los bits 15 a 8) y PCLATU (que contiene en sus 5 primeros valores binarios, los bits 20 al 16), que son los medios para actualizar el valor del PC o de los registros PCU, PCH y PCL. Los registros PCLATCH, PCLATU y PCL pueden ser manipulados por el programador.

Cuando se quiere actualizar el registro PC con un valor, se debe colocar el nuevo código binario en los registros PCLATU y PCLATH. A pesar de que se ha escrito una nueva cifra en ambos registros, el PC no actualizará su estado con estos valores hasta que se haya realizado una escritura sobre el registro PCL. El PCL es el único registro que se actualiza automáticamente con la información del PC. Esta propiedad permite la elaboración de tablas de datos en la memoria de programa, tal como se puede observar en el ejemplo que se muestra en el programa 4.9.

Programa 4.9. Manejo de una tabla de datos haciendo uso del Contador de programa (PC)

```
        org 0x00            ;Directiva que establece el punto de inicio del programa en el
                            ;vector de Reset.
        goto MAIN           ;La instrucción goto MAIN va dentro del vector de Reset de la
                            ;memoria FLASH.
        org 0x200           ;Se establece el inicio del programa en la dirección 0x200 de la
                            ;FLASH.
MAIN:
        clrf 0x20,0         ;La dirección absoluta 0x20 de la memoria SRAM se pone a 0.
        lfsr FSR0,0x100     ;Se carga la dirección 0x100 en el puntero FSR0.
        movlw 0x02          ;Se carga el valor 0x02 en el registro W.
        movwf PCLATH        ;Se pone este valor en el PCLATH (dado que el programa
                            ;comienza en 0x200).
INICIO:
        movf 0x20,W,0       ;Se carga el valor almacenado en la dirección absoluta 0x20
                            ;de la SRAM en W
        call TABLA          ;Se  salta a la dirección especificada por la etiqueta TABLA
        movwf POSTINC0      ;Se agrega el valor retornado en W a la dirección apuntada
                            ;por FSR0
        movlw .2            ;Se carga el valor de .2 al registro W
        addwf 0x20,f,0      ;Se suma el valor almacenado en la dirección absoluta 0x20
                            ;con 2
        movlw .8            ;Se carga el valor decimal .8 en el registro W
        cpfseq 0x20,0       ;Se compara el contenido de la dirección 0x20 con .8 y si son
                            ;iguales habrá un salto
        goto  INICIO        ;De ser diferentes los valores se retorna a INICIO y se repite
                            ;el proceso
FIN:
        goto  FIN           ;Fin del programa en el bucle infinito goto FIN
TABLA:
        addwf PCL,f         ;El valor del registro W se suma al PCL y el PC toma el
                            ;nuevo valor
        retlw 'H'           ;Se retorna de la instrucción call con W = ASCII ‘H’
        retlw 'o'           ;Se retorna de la instrucción call con W = ASCII ‘o’
        retlw 'l'           ;Se retorna de la instrucción call con W = ASCII ‘l’
        retlw 'a'           ;Se retorna de la instrucción call con W = ASCII ‘a’
```

Como se puede observar en el análisis del programa, el código comienza en la dirección 0x200 de la memoria FLASH. A partir de este punto, el registro PCL toma el valor 0x00 (por ser el byte bajo de la dirección del PC), PCH toma el valor 0x02 y PCU 0x00. Por otro lado, manualmente se debe cargar el

registro PCLATH con 0x02, ya que en la instrucción addwf PCL,f carga el registro PC con los valores de los registros PCLATU y PCLATH que por defecto son 0. Por esta razón, de no usar la instrucción movwf PCLATH con W = 0x02, se cargará el PC con el valor 0x0000+PCL que corresponde a una dirección fuera del rango del programa, lo que ocasionaría que el código se reinicie.

En el gráfico 4.9 se muestra un esquema de la configuración de la memoria de datos y de la memoria de programa una vez finalizado el programa 4.9. La instrucción addwf PCL,f se encuentra almacenada en la dirección 0x222 de la memoria FLASH. Cuando se suma un valor al PCL, este debe ser par, ya que las direcciones de la memoria de programa son siempre valores pares. La primera vez, cuando se suma el valor 0 al PCL, no ocurre ningún cambio aparente en el PC, por lo cual se autoincrementa en 2 y se ejecuta la instrucción retlw 'H' (en la dirección 0x224). Cuando el registro W entra a la rutina TABLA con el valor de 0x02, el PCL se incrementa en 2 más su autoincremento, tomando un valor de incremento de 4, con lo cual se ejecuta la instrucción retlw 'o' (en la dirección 0x226). Conforme se incrementa en 2 el valor del registro W, la sumatoria del PCL permitirá acceder a las instrucciones superiores.

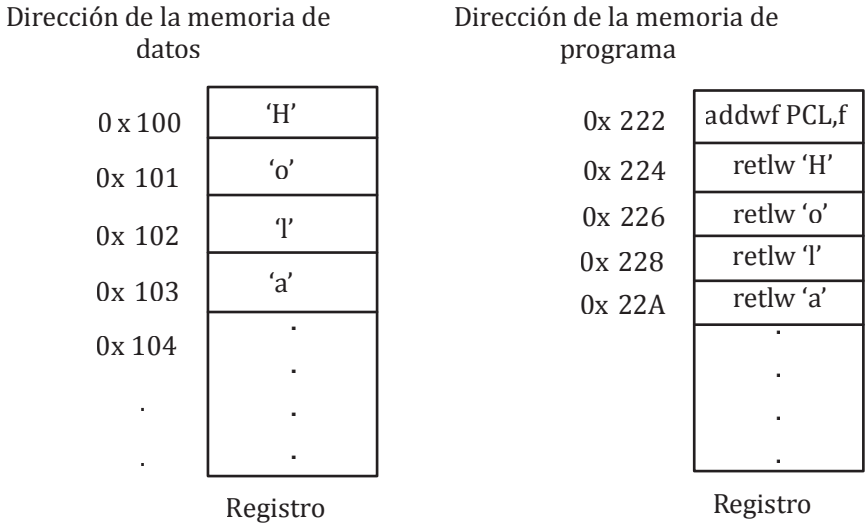
La parte de almacenamiento de la memoria de datos haciendo uso del modo de direccionamiento indirecto es conocida. El tema crucial es incrementar la variable que se pasará al registro W luego del llamado a la etiqueta TABLA de dos en dos. Esto se logra con las instrucciones movlw .2 y addwf 0x20,f,0, las cuales permiten que el valor almacenado en la dirección absoluta 0x20 aumente de a dos. De esta manera, por cada llamado a la etiqueta TABLA se ejecutará una instrucción retlw diferente. Cabe señalar que el programador debe comparar el resultado de la dirección absoluta 0x20 con el máximo número de instrucciones retlw, para evitar así que el PCL tome un valor por encima de la última instrucción: retlw 'a' (para esto, se compara el valor de la dirección 0x20 con .8, ya que este es el valor que tomará la variable al retornar *retlw 'a'*).

Uno de los inconvenientes de esta rutina es el uso de un contador de direcciones pares y el hecho de que el programador tenga que tener cuidado con enviar a la rutina TABLA un valor de W mayor al número de instrucciones retlw. El otro problema es que por cada instrucción retlw se está desperdiciando una dirección de 16 bits, cuando el valor de retorno es de 8 bits. Esto genera un desperdicio de la capacidad de memoria en un 50%. Existe otro método que permite mejorar este aspecto. Esto se verá en el subcapítulo 4.5.

4.4 La pila de direcciones

La pila de direcciones es un bloque de memoria SRAM compuesto de 31 direcciones de 21 bits que trabaja directamente relacionado con el PC para el almacenamiento de los valores de este registro durante las instrucciones de salto que tiene la opción de retorno, como es el caso de las instrucciones *call*, *rcall* y las interrupciones. La pila se compone de un puntero de pila o registro de nombre STKPTR de 8 bits, el cual apunta a la dirección de la pila donde se ha almacenado el último valor del PC. Este registro utiliza sus 5 bits menos significativos para apuntar al valor de la pila (cifras de 0 a 31) y los dos bits más significativos como banderas de estado de pila. El registro TOS de 21 bits posee el contenido de la pila y se encuentra compuesto por tres registros completamente accesibles: TOSU (bits 20 al 16), TOSH (bits 15 al 8) y TOSL (bits 7 al 0). Para comprender más claramente el funcionamiento de la pila se debe analizar el programa 4.10.

Gráfico 4.9. Arreglo de la memoria de datos y de la memoria FLASH en el programa 4.9



Elaboración propia.

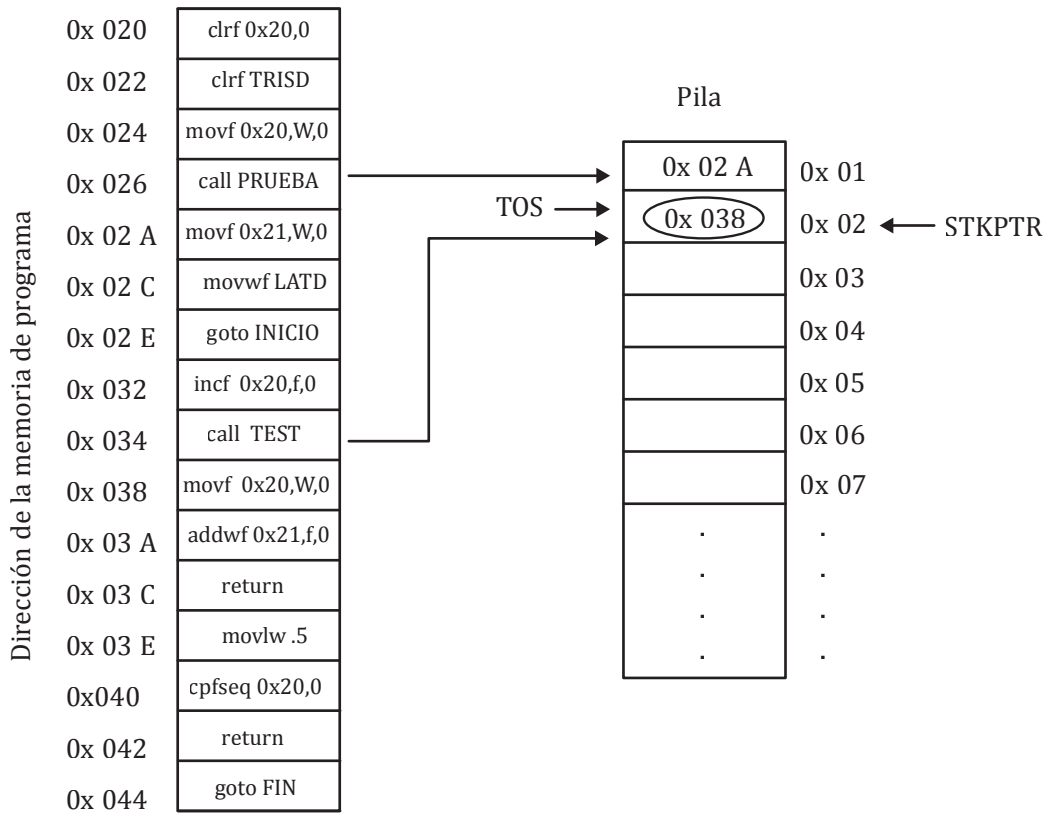
Programa 4.10. Manejo de la pila en instrucciones de salto

```
        Org 0x00
        goto MAIN
        org 0x020
MAIN:
        clrf 0x20,0             ;0x020
        clrf TRISD              ;0x022
INICIO:
        movf 0x20,W,0           ;0x024
        call PRUEBA             ;0x026-0x028
        movf 0x21,W,0           ;0x02A
        movwf LATD              ;0x02C
        goto INICIO             ;0x02E-0x030
PRUEBA:
        incf  0x20,f,0          ;0x032
        call  TEST              ;0x034-0x36
        movf  0x20,W,0          ;0x038
        addwf 0x21,f,0          ;0x03A
        return                  ;0x03C
TEST:
        movlw .5                ;0x03E
        cpfseq 0x20,0           ;0x040
        return                  ;0x042
FIN:
        goto FIN                ;0x044-0x046
```

En el programa 4.10 se muestran comentadas al lado derecho las direcciones de la memoria de programa donde se encuentra almacenada cada instrucción. Si se observa la instrucción call PRUEBA, esta se encuentra en la dirección 0x026 de la memoria de programa. En el momento en que se ejecuta dicha instrucción, el PC se carga con el valor de la etiqueta PRUEBA: 0x032. Sin embargo, durante la ejecución de la instrucción, la dirección contigua 0x02A se almacena en la pila, de manera que el puntero de pila STKPTR toma el valor de 1.

Posteriormente, se ejecuta la instrucción incf 0x20,f,0, y luego se ejecuta la instrucción de salto call TEST. En este momento se guarda en la pila el valor contiguo, que es 0x038, incrementándose el valor del puntero de pila STKPTR a 2. Luego, el PC toma el valor de 0x03E. Este esquema se observa en el gráfico 4.10.

Gráfico 4.10. Almacenamiento de las direcciones posteriores a las instrucciones de salto en la pila



Elaboración propia.

Durante las instrucciones de retorno (*Return*) ocurre el caso inverso. El puntero de pila se resta en uno, pasando el contenido del registro TOS al PC. Con la ejecución del programa se retorna a la instrucción siguiente al último salto realizado. Lo lógico es que el programador intente mantener un número de instrucciones *Return* igual al número de instrucciones de salto, ya que la pila mantiene los valores en memoria y se podría tomar una dirección no deseada.

Para la manipulación directa de la pila existen dos instrucciones: PUSH y POP. La primera instrucción almacena el valor siguiente que tomará el PC en la pila, incrementando el puntero STKPTR en 1 (similar al efecto de una instrucción *call* o *rcall*). La segunda instrucción copia el contenido del registro TOS en el PC y luego resta el STKPTR en 1 (similar al efecto de una instrucción *Return*, *retlw* o *retfie*).

El programa 4.11 es prácticamente igual al programa 4.10, pero se ha agregado una instrucción PUSH en la dirección 0x03E para poder observar las diferencias.

Programa 4.11. Ejemplo de funcionamiento de la instrucción PUSH

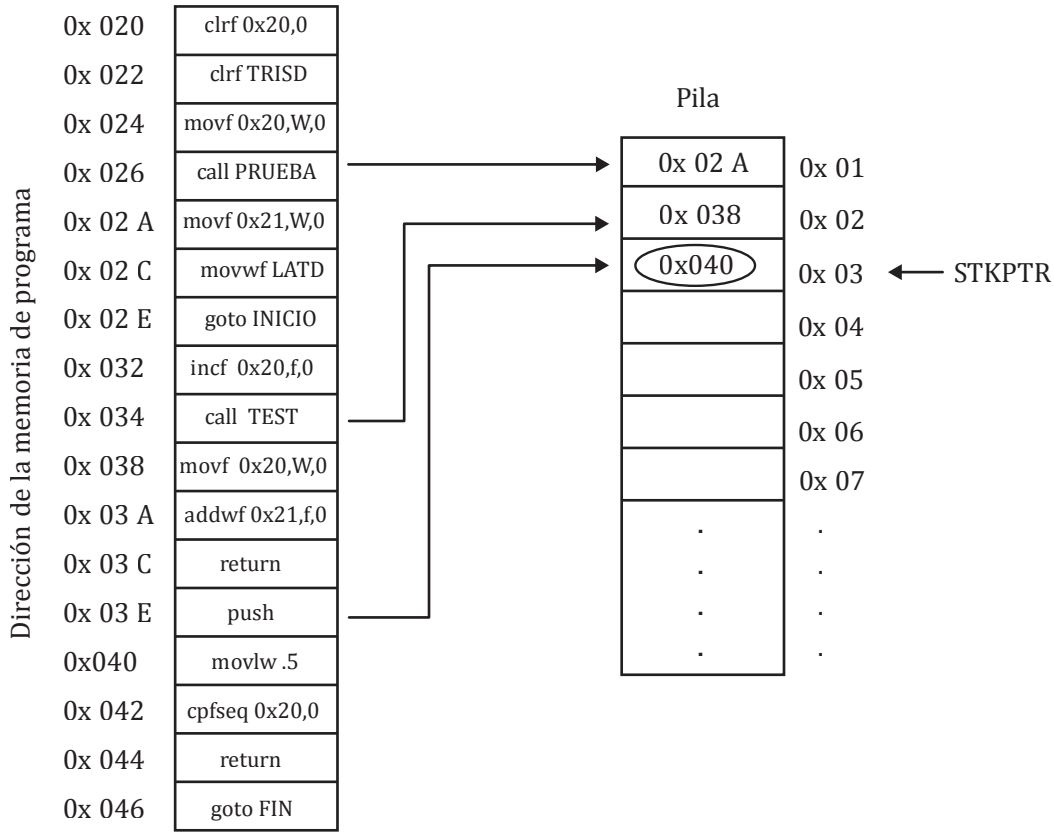
```
        org 0x00
        goto MAIN
        org 0x020
MAIN:
        clrf 0x20,0             ;0x020
        clrf TRISD              ;0x022
INICIO:
        movf 0x20,W,0           ;0x024
        call PRUEBA             ;0x026
        movf 0x21,W,0           ;0x02A
        movwf LATD              ;0x02C
        goto INICIO             ;0x02E
PRUEBA:
        incf  0x20,f,0          ;0x032
        call  TEST              ;0x034
        movf  0x20,W,0          ;0x038
        addwf 0x21,f,0          ;0x03A
        return                  ;0x03C
TEST:
        push                    ;0x03E
        movlw .5                ;0x040
        cpfseq 0x20,0           ;0x042
        return                  ;0x044
FIN:
        goto FIN                ;0x046
```

Analizando el programa 4.11 se puede observar que hasta la segunda instrucción *call* no ha ocurrido ningún cambio aparente en el programa. El registro TOS contiene el valor 0x38 y el STKPTR apunta a la dirección 2. Cuando en la dirección 0x03E se ejecuta con la instrucción PUSH, el STKPTR se incrementa en 1, apuntando en la dirección 3, y el contenido de esta toma el valor de 0x040 (registro TOS). La dirección 0x040 de la memoria FLASH contiene a la instrucción movlw .5. Después de ejecutar el PUSH, el PC se continua incrementando de forma normal ejecutando las instrucciones de las direcciones 0x040, 0x042 y 0x044.

Es en la instrucción *Return* de la dirección 0x044 que ocurre un cambio en el programa. Esta instrucción debería llevar al PC a la dirección 0x038. Sin embargo, el STKPTR apunta a la dirección 3, donde el contenido del TOS es 0x040. Por tanto, el *Return* generará que el PC se cargue con el valor 0x040, ejecutándose nuevamente las tres instrucciones de las direcciones 0x040, 0x042 y 0x044. En

esta última, al volverse a ejecutar la instrucción *Return* por segunda vez, se carga en el PC el valor 0x038 (ya que el STKPTR se restó en 1) y el programa seguirá su ciclo normal. En el gráfico 4.11 se observa el efecto de haber usado la instrucción PUSH en el programa.

Gráfico 4.11. Efecto de utilización de la instrucción PUSH



Elaboración propia.

Si se quisiera dejar sin efecto a la instrucción PUSH, simplemente se agrega una instrucción POP que realiza el proceso opuesto, es decir, resta el STKPTR en 1. Con lo cual, al ejecutar el primer *Return*, el PC tomará el valor 0x038, manteniendo el esquema de ejecución del programa 4.10.

Un detalle para tomar en cuenta es que la pila cuenta con 31 registros para almacenamiento de los diferentes valores del PC en instrucciones de salto o para el uso de las instrucciones PUSH y POP. El programador debe evitar llenar la pila con más de 31 valores (es decir, incrementar STKPTR por encima de 31), y tampoco debe restar STKPTR por debajo de 1. De ocurrir esta situación, el registro STKPTR cuenta con banderas (bit 7: STKFUL y bit 6: STKUNF) que indican cuando un evento de sobrellenado ha ocurrido (STKFUL = '1') o un evento de sobrevaciado ha ocurrido (STKUNF = '1'). Estas condiciones producen una condición de Reset en el microcontrolador, por lo cual deben ser evitadas.

Adicionalmente, existe una pila rápida de 3 bytes que permite almacenar automáticamente el valor del BSR, el registro W y el registro STATUS. Esta pila se invoca cuando se realiza la instrucción call dir,1, donde el segundo operador indica que se deben apilar los registros en mención. Para recuperar el valor de los registros después de la operación de retorno se deberá utilizar la instrucción return 1, donde el operador 1 indica la recuperación de los valores almacenados en la pila rápida.

4.5 Instrucciones para el manejo de tablas en la memoria de programa

Para mejorar el tema del desperdicio de memoria de programa para almacenamiento de datos y simpleza en el manejo de las direcciones de programa, se pueden usar los registros punteros de tabla. Estos registros trabajan con las siguientes instrucciones especiales:

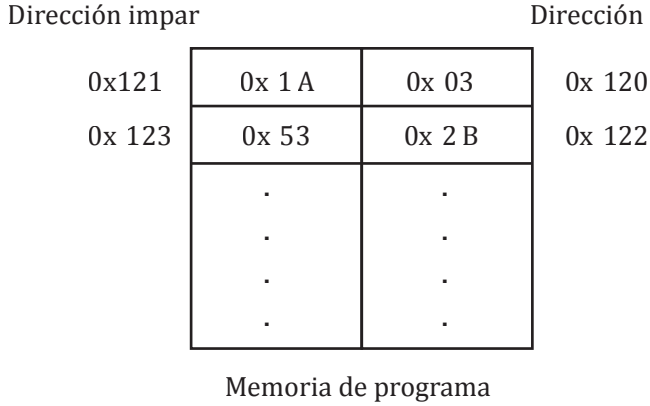
TBLRD*	(Lectura de tabla)
TBLRD*+	(Lectura de tabla con incremento posterior del puntero de tabla)
TBLRD*-	(Lectura de tabla con decremento posterior del puntero de tabla)
TBLRD+*	(Lectura de tabla con incremento anterior del puntero de tabla)
TBLWT*	(Escritura de tabla)
TBLWT*+	(Escritura de tabla con incremento posterior del puntero de tabla)
TBLWT*-	(Escritura de tabla con decremento posterior del puntero de tabla)
TBLWT+*	(Escritura de tabla con incremento anterior del puntero de tabla)

Para definir una tabla de datos se tiene que hacer uso de algunas directivas, como org (para indicar el punto de inicio donde estará almacenada la tabla en la memoria de programa) y db (para establecer los bytes que estarán almacenados en el arreglo). Por ejemplo, para definir una tabla de 4 bytes que ocupe las direcciones 0x120, 0x121, 0x122 y 0x123 se ejecutan las siguientes directivas:

```
1. org 0x120
2. TABLA: db 0x03, 0x1A, 0x2B, 0x53
```

Estas directivas permiten almacenar cuatro bytes ocupando las direcciones 0x120 a 0x123 de la memoria de programa. Las direcciones impares (0x121 y 0x123) representan el byte más significativo de la dirección par de 16 bits. En el gráfico 4.12 se observa el esquema de los datos de la tabla almacenados en la memoria de programa. La etiqueta TABLA permite ubicar la dirección donde se inicia el bloque de datos.

Gráfico 4.12. Esquema de los bytes de la tabla creada almacenados en la memoria de programa



Elaboración propia.

Una vez creada una tabla de datos, es posible hacer uso de esta mediante los registros de tabla. El registro puntero de tabla TBLPTR posee 21 bits y apunta a la dirección de la memoria de programa donde se quiere realizar la lectura de un byte. A diferencia del registro PC, el TBLPTR puede tomar valores impares, con lo cual puede acceder a la parte alta de la palabra de 16 bits de la memoria de programa como si fuera una dirección impar.

El registro TBLPTR se encuentra dividido en tres registros de 8 bits: TBLPTRU (bits 20 al 16), TBLPTRH (bits 15 al 8) y TBLPTRL (bits 7 al 0). Cuando se requiere acceder a una dirección de la memoria de programa se debe escribir en estos tres registros. El registro TABLAT es de 8 bits de longitud y contiene el valor almacenado en la dirección establecida por TBLPTR. Para descargar el contenido de la dirección en el registro TBLPTR se debe ejecutar la instrucción TBLRD*.

Para cargar la dirección de la tabla de datos de 21 bits es necesario realizar tres lecturas de la dirección (una para los 5 bits más significativos y dos para los 16 bits restantes). Esto se logra con los operadores UPPER, HIGH y LOW. Por ejemplo, se crea la siguiente tabla:

```
org 0x011267
MENSAJE: db 'a','b','c','d'
```

Esto implica que la etiqueta MENSAJE es equivalente al valor 0x011267, que es de 21 bits. Para poder cargar esta dirección en el registro TBLPTR se requiere la siguiente secuencia de instrucciones:

```
1. movlw UPPER MENSAJE     ;Se carga en W el valor 0x01
2. movwf TBLPTRU           ;Se ingresan los 5 bits más significativos de
                           ;TBLPTR
3. movlw HIGH MENSAJE      ;Se carga en W el valor 0x12
4. movwf TBLPTRH           ;Se ingresa en TBLPTR el valor correspondiente a
                           ;los bits 15 al 8
5. movlw LOW MENSAJE       ;Se carga en W el valor 0x67
6. movwf TBLPTRL           ;Se ingresa el byte menos significativo en
                           ;TBLPTR.
```

Antes de hacer uso de la tabla, es necesario configurar el registro de función especial EECON1 del PIC18F, a fin de seleccionar la memoria FLASH para el acceso mediante el puntero de tabla. En el gráfico 4.13 se muestran los bits de configuración de dicho registro.

Gráfico 4.13. Registro EECON1

Elaboración propia.

Los bits del registro EECON1 son los siguientes:

EEPGD. Es el bit de selección de acceso a la memoria FLASH o EEPROM del microcontrolador. Para hacer uso de la tabla es necesario seleccionar la memoria FLASH poniendo este bit a estado '1'.

CFGS. Este es un bit de selección general de acceso a la memoria de programa y EEPROM o a los registros de configuración de la memoria de programa. Por defecto se encuentra en nivel '0', seleccionando la primera opción.

FREE. Es el bit habilitador del borrado de una fila. Si este bit se encuentra en nivel '1' se habilita el borrado de las direcciones apuntadas por TBLPTR. En caso contrario, solo se permiten operaciones de escritura.

WREER. Bit de bandera de error de escritura de memoria de programa o de memoria EEPROM. Si este bit es '1', ha ocurrido un error de escritura. Si el bit es '0', la escritura ha sido satisfactoria.

WREN. Este bit habilita la escritura en la memoria de programa o EEPROM. Si es 1_2, la escritura se encuentra habilitada. En caso contrario, la escritura queda inhibida.

WR. Este bit solo puede ponerse a '1'. Cuando esto se hace, se inicia un ciclo de escritura o borrado de la memoria de programa o EEPROM. El bit se pone automáticamente a '0' para indicar que el proceso ha finalizado.

RD. Bit de control de lectura. Si este bit se pone a '1', se inicia una lectura de la memoria EEPROM.

En el programa 4.12 se muestra un ejemplo de una lectura de tabla de 4 bytes. Estos valores leídos de la tabla se envían a las direcciones absolutas 0x100, 0x101, 0x102 y 0x103 de la memoria de datos.

Programa 4.12. Ejemplo de lectura de un bloque de datos en la memoria de programa haciendo uso de los registros de manejo de tabla

```
        org 0x00              ;Dirección del vector de Reset.
        goto MAIN             ;Primera instrucción salto a la dirección 0x200 de la memoria
                              ;de programa.
        org 0x120             ;Se establece la dirección 0x120 de la memoria de programa.
TABLA: db  'H','o','l','a'    ;Se crean 4 bytes en las direcciones 0x120, 0x121, 0x122 y
                              ;0x123.
        org 0x200             ;Se establece la dirección 0x200 de la memoria de programa.
MAIN:
        lfsr  FSR0,0x100      ;Se carga el registro FSR0 con la dirección 0x100 de la
                              ;memoria de datos.
        bsf   EECON1,EEPGD    ;Se selecciona la memoria FLASH para el acceso con
                              ;los registros de tabla.
        bcf   EECON1,CFGS     ;Se selecciona el modo de acceso a la memoria FLASH
                              ;y EEPROM.
        movlw UPPER TABLA     ;Se leen los 5 bits más significativos de la dirección
                              ;de la tabla.
        movwf TBLPTRU         ;Se ingresan al registro alto del puntero de tabla.
        movlw HIGH TABLA      ;Se leen los bits 15 al 8 de la dirección de la tabla.
        movwf TBLPTRH         ;Se ingresan al registro TBLPTRH del puntero de tabla.
```

```
        movlw LOW TABLA     ;Se leen los 8 bits menos significativos de la tabla.
        movwf TBLPTRL       ;Se cargan estos 8 bits en el registro TBLPTRL,
                            ;quedando configurado TBLPTR.
INICIO:
        TBLRD*              ;Se carga el contenido de la dirección apuntada por
                            ;TBLPTR en TABLAT.
        movf TABLAT,W       ;Se copia el contenido de TABLAT al registro W.
        movwf INDF0         ;Se carga el contenido leído de la Tabla en INDF0.
        incf FSR0L          ;Se incrementa el puntero de direcciones de la memoria de
                            ;datos.
        incf TBLPTRL,f,0    ;Se incrementa la parte baja del puntero de tabla para leer
                            ;el siguiente byte.
        movlw 0x24          ;Se carga el registro W con el valor 0x24 (última dirección
                            ;de TBLPTRL)
        cpfseq TBLPTRL,0    ;Se compara TBLPTRL con 0x24, si son iguales se
                            ;termina la lectura de la tabla.
        goto  INICIO        ;Si son diferentes se repite el programa leyendo la
                            ;siguiente dirección de la tabla.
FIN:
        goto FIN            ;Se repite el proceso de forma indefinida.
```

El incremento del registro TBLPTRL permite apuntar a la siguiente dirección de la tabla. Sin embargo, existe otra forma para poder tener acceso a los bytes de la tabla sin tener que incrementar el registro TBLPTRL. Esto se logra con la instrucción TLBLRD*+. Este comando equivale a realizar los siguientes procesos:

1. TBLRD*
2. incf TBLPTRL,f,0

Por ejemplo, el programa 4.13 es una versión equivalente al programa 4.12 haciendo uso de la instrucción TLBRD*+.

Programa 4.13. Ejemplo de lectura de un bloque de datos en la memoria de programa haciendo uso de la instrucción TLBRD*+

```
        org 0x00                ;Dirección del vector de Reset.
        goto MAIN               ;Primera instrucción salto a la dirección 0x200 de la memoria
                                ;de programa.
        org 0x120               ;Se establece la dirección 0x120 de la memoria de programa.
TABLA: db  'H','o','l','a'      ;Se crean 4 bytes en las direcciones 0x120, 0x121, 0x122 y
                                ;0x123.
        org 0x200               ;Se establece la dirección 0x200 de la memoria de programa.
MAIN:
        lfsr  FSR0,0x100        ;Se carga el registro FSR0 con la dirección 0x100 de la
                                ;memoria de datos
        bsf   EECON1,EEPGD      ;Se selecciona la memoria FLASH para el acceso con
                                ;los registros de tabla.
        bcf   EECON1,CFGS       ;Se selecciona el modo de acceso a la memoria FLASH
                                ;y EEPROM.
        movlw UPPER TABLA       ;Se leen los 5 bits más significativos de la dirección de
                                ;la tabla.
        movwf TBLPTRU           ;Se ingresan al registro alto del puntero de tabla.
        movlw HIGH TABLA        ;Se leen los bits 15 al 8 de la dirección de la tabla.
        movwf TBLPTRH           ;Se ingresan al registro TBLPTRH del puntero de tabla.
        movlw LOW TABLA         ;Se leen los 8 bits menos significativos de la tabla.
        movwf TBLPTR            ;Se cargan estos 8 bits en el registro TBLPTRL, quedando
                                ;configurado TBLPTR.
INICIO:
        TBLRD*+                 ;Se carga el contenido apuntado por TBLPTR en TABLAT
                                ;y se incrementa TBLPTR en 1.
        movf TABLAT,W           ;Se copia el contenido de TABLAT al registro W.
        movwf INDF0             ;Se carga el contenido leído de la Tabla en INDF0.
        incf FSR0L              ;Se incrementa el puntero de direcciones de la memoria de
                                ;datos.
        movlw 0x24              ;Se carga el registro W con el valor 0x24 (última dirección
                                ;de TBLPTRL)
        cpfseq TBLPTRL,0        ;Se compara TBLPTRL con 0x24, si son iguales se
                                ;termina la lectura de la tabla.
        goto  INICIO            ;Si son diferentes se repite el programa leyendo la siguiente
                                ;dirección de la tabla.
FIN:
        goto FIN                ;Se ingresa a un bucle indefinido.
```

4.6 Acceso a la memoria EEPROM de datos

La memoria EEPROM de datos es una unidad de almacenamiento que muchos microcontroladores PIC18F poseen para las operaciones de lectura y escritura no volátiles. Esta memoria se encuentra muy aparte de la memoria de datos y de programa, manejando su bus de datos y direcciones de forma independiente. La memoria EEPROM para el PIC18F4550 posee una capacidad de 256 bytes (bus de direcciones de 8 bits) con una longitud de 1 byte por dirección.

Uno de los inconvenientes de las memorias EEPROM es que sus tiempos de escritura son lentos (demoran alrededor de 4 milisegundos), mientras que las lecturas se pueden realizar a altas velocidades. La memoria EEPROM del PIC18F tiene un ciclo de duración de lectura/escritura de hasta 1 millón de accesos.

Para acceder a la memoria EEPROM se requiere hacer uso de cuatro registros de función específica: EECON1, EECON2, EEDATA (bus de datos) y EEADR (bus de direcciones). El registro EECON2 es de tipo virtual, quiere decir que físicamente no se encuentra implementado en el mapa de memoria de los RFE.

El inicio del proceso de escritura de la EPROM requiere los siguientes pasos:

1. Configurar los bits EEPGD y CFGS ambos en '0', para que se seleccione el acceso a la memoria EEPROM.
2. Colocar la dirección a acceder en el registro EEADR.
3. Cargar el dato a grabar en el registro EEDATA.
4. Poner a nivel '1' el bit WREN para habilitar la opción de escritura.
5. De haber interrupciones activas, estas deben deshabilitarse colocando el bit GIE del registro INTCON a nivel '0' lógico. El motivo de este paso es evitar que la secuencia de bytes que posteriormente se van a enviar al registro EECON2 sea interrumpida por otro proceso.
6. Cargar el valor 0x55 e inmediatamente el valor 0xAA en el registro EECON2.
7. Luego, se pone a nivel '1' el bit WR, a fin de iniciar el proceso de escritura.
8. Se espera a que este bit se ponga automáticamente en nivel '0'. Esto puede tardar alrededor de 4 milisegundos.
9. Se reactivan las interrupciones en caso se requieran y se desactiva la escritura poniendo a '0' el bit WREN.

En el programa 4.14 se muestra un ejemplo de una rutina de escritura de la memoria EEPROM. Esta rutina escribe el código ASCII 'A' en la dirección 0x10 de la memoria. En este ejemplo no se considera la desactivación y reactivación de la máscara de interrupción, ya que corresponden a un tema que aún no se ha tocado y se verá en el capítulo 6.

Programa 4.14. Ejemplo de escritura de un dato en la memoria EEPROM

```
ESCRITURA:
        bcf EECON1, CFGS        ;Se habilita el acceso a memoria EEROM y Flash
        bcf EECON1, EEPGD       ;Se selecciona el acceso a memoria EEPROM
        movlw 0x10              ;Se selecciona la dirección 0x10 de la memoria EEPROM
        movwf EEADR             ;y se carga en el registro EEADR.
        movlw 'A'               ;Se carga el valor ASCII 'A' en el registro W
        movwf EEDATA            ;y se carga en EEDATA para su almacenamiento en la
                                ;dirección 0x10 de la EEPROM.
        bsf EECON1,WREN         ;Se habilita el proceso de escritura.
        movlw 0x55              ;Se carga el valor 0x55 en el registro W.
        movwf EECON2            ;Se envía al registro EECON2 tal como lo establece la
                                ;hoja de datos del PIC18F.
        movlw 0xAA              ;Se carga el valor 0xAA en el registro W.
        movwf EECON2            ;Se envía al registro EECON2 tal como lo establece la
                                ;hoja de datos del PIC18F.
        bsf   EECON1, WR        ;Se inicia el proceso de escritura
ESPERA:
        btfsc EECON1,WR         ;Se espera a que el bit WR automáticamente tome el valor
                                ;'0' lógico
        goto  ESPERA            ;Se salta a ESPERA mientras el bit WR siga siendo '1'.
        bcf EECON1,WREN         ;Se  culmina el proceso  y se deshabilita la escritura en
                                ;la EEPROM.
        Return
```

Un detalle para tomar en cuenta en la rutina mostrada en el programa 4.14 es que el bucle ESPERA va a tardar alrededor de 4 milisegundos, que es el tiempo mencionado que la memoria EEPROM tarda en escribir un dato.

El proceso de lectura es un poco más sencillo que el de escritura, ya que se deben realizar menos pasos. Estos son los siguientes:

1. Configurar los bits EEPGD y CFGS ambos en '0' para que se seleccione el acceso a la memoria EEPROM.
2. Establecer la dirección de lectura colocando el valor en el registro EEADR
3. Poner a nivel '1' lógico el bit RD. Luego de esto, en un solo ciclo de instrucción el dato estará listo para su lectura. El bit RD se pondrá automáticamente a nivel '0' para indicar esta situación.
4. Leer el valor del registro EEDATA.

En el programa 4.15 se muestra un ejemplo de una rutina de lectura de la memoria EEPROM. Esta rutina lee la misma dirección que se usó en la rutina de escritura y devuelve en el registro W el valor leído.

Programa 4.15. Ejemplo de lectura de un dato en la memoria EEPROM

```
LECTURA:
        bcf   EECON1,CFGS       ;Se habilita el acceso a memoria EEROM y Flash.
        bcf   EECON1,EEPGD      ;Se selecciona el acceso a memoria EEPROM.
        movlw 0x10              ;Se selecciona la dirección de lectura 0x10
        movwf EEADR             ;y se carga en el registro EEADR.
        bsf   EECON1, RD        ;Se activa el comando de lectura
        movf  EEDATA, W         ;Un ciclo de instrucción después se puede leer el
                                ;valor en el registro W.
        return                  ;Se retorna con el registro W cargando el dato leído.
```

4.7 Directivas para el lenguaje ensamblador

Las directivas son sentencias que parecen instrucciones, ya que se mezclan con el lenguaje ensamblador. Pero, en realidad cada directiva no se traduce en un código máquina, sino que más bien representa información para que el compilador interprete cómo debe organizar el lenguaje máquina en la memoria de programa. Las directivas se caracterizan por su función. Existen en total cuatro clases de directivas:

1. Directivas de control
2. Directivas de datos
3. Directivas informativas
4. Macros

Las directivas de control se usan frecuentemente para indicar bloques de código a compilar, definir sectores de memoria y realizar la ejecución repetida de un sector del programa. Algunas de estas directivas ayudan a crear diferentes versiones de código, con lo cual se puede ver la evolución de un algoritmo conforme se realizan cambios. En el gráfico 4.14 se muestran las principales directivas de control.

Gráfico 4.14. Directivas de control del compilador MPASM de MPLAB para PIC18F

Directiva	Función
#DEFINE	Crea un alias a un texto del programa.
#undefine	Elimina el alias previamente creado.
IF	Condicional que evalúa una condición que siendo verdadera permite compilar una porción de código.
ELSE	Condicional negado de la condición en If.
ENDIF	Fin del condicional If.
IFDEF	Condicional que evalúa la existencia de un alias.
IFNDEF	Condicional que evalúa la no existencia de un alias.
While	Permite iniciar un bucle iterativo.

Directiva	Función
Endw	Fin del bucle While.
CODE	Define un sector de memoria.
#INCLUDE	Permite incluir archivos externos al programa.
RADIX	Permite el cambio de base numérica.
END	Indica el fin del programa.

Elaboración propia.

1. #DEFINE <alias> <texto>
   Esta directiva permite reemplazar un texto por un alias. Es posible definir una constante, como también una instrucción, para luego invocar al alias en el programa. Las directivas #define deben ir fuera del segmento de código del programa. En el programa 4.16 se muestra un ejemplo de uso de esta directiva.

Programa 4.16. Ejemplo de uso de la directiva #DEFINE

```
#define SUMA_10 addlw .10      ;El texto SUMA_10 es un alias a la instrucción addlw
                               ;.10
#define PESO .5                ;El texto PESO es un alias al valor constante 5
        org 0x00               ;Vector de Reset
        goto MAIN              ;Se salta a la dirección 0x200 de la memoria de programa
        org 0x200
MAIN:
        clrf 0x20,0            ;El registro apuntado por la dirección 0x20 de la SRAM
                               ;se pone a 0.
        movlw PESO             ;Se carga el valor .5 en el registro W
        SUMA_10                ;Se ejecuta la instrucción addlw .10
        SUMA_10                ;Se vuelve a ejecutar la instrucción addlw .10
```

2. #UNDEFINE <alias> <texto>
   La directiva #UNDEFINE deshace los alias previamente creados por la directiva #DEFINE. Esta operación es necesaria cuando se utilizan múltiples módulos en el proyecto y se desconoce si algún alias ya ha sido usado en alguna parte del programa. Para evitar la reutilización, se puede deshacer el alias y volverlo a definir. Se debe tener mucho cuidado al utilizar esta directiva, ya que, de no existir el alias, el compilador generará un error. En el programa 4.17 se muestra un ejemplo de uso de esta directiva.

Programa 4.17. Ejemplo de uso de la directiva #UNDEFINE

```
#define SUMA_10 addlw .10      ;El texto SUMA_10 es un alias a la instrucción addlw .10.
#define PESO  .5               ;El texto PESO es un alias al valor constante 5
        org 0x00               ;Vector de Reset
        goto MAIN              ;Se salta a la dirección 0x200 de la memoria de programa
        org 0x200
#undefine PESO                 ;Se elimina el alias PESO
#define   PESO .2              ;Se vuelve a crear el alias PESO para la constante .2
MAIN:
        clrf 0x20,0            ;El registro apuntado por la dirección 0x20 de la SRAM
                               ;se pone a 0
        movlw PESO             ;Se carga el valor .2 en el registro W
        SUMA_10                ;Se ejecuta la instrucción addlw .10
        SUMA_10                ;Se vuelve a ejecutar la instrucción addlw .10
```

3. IF <condición> - ELSE-ENDIF

   Las sentencias IF-ELSE-ENDIF trabajan juntas para definir bloques de código que serán ejecutados siempre que una condición sea verdadera o falsa. La condición es verdadera cuando un alias tiene un valor definido y se realiza una comparación con dicho valor. En caso contrario, la condición es falsa. El código de programa se puede incluir entre las sentencias IF – ELSE, compilando dicho código cuando la condición es verdadera. Mientras que el bloque de código entre ELSE – ENDIF se compilará si la condición es falsa. De esta manera, pueden existir dos versiones de un programa o de porciones del programa en un mismo archivo de texto. En el programa 4.18 se muestra un ejemplo de uso de estas directivas.

Programa 4.18. Ejemplo de uso de las directivas IF – ELSE – ENDIF

```
#define SUMA_10 addlw .10      ;El texto SUMA_10 es un alias a la instrucción addlw .10.
#define VERSION  2             ;El texto VERSION es un alias a la constant 2
        org 0x00               ;Vector de Reset
        goto MAIN              ;Se salta a la dirección 0x200 de la memoria de programa
        org 0x200
MAIN:
        IF(VERSION == 2)       ;Si VERSION es alias de 2, la condición es verdadera.
        clrf 0x20,0            ;Entonces se compilan las instrucciones entre el IF-ELSE
        movlw .5
        SUMA_10
        SUMA_10
        ELSE                   ;La condición falsa no se ejecuta y por tanto las
                               ;instrucciones comprendidas
        setf 0x20,0            ;entre ELSE y ENDIF no se compilan
        movlw .3
        SUMA_10
        ENDIF
```

Cabe señalar que la directiva ELSE no es obligatoria. Es posible eliminar del programa 4.18 la directiva ELSE y las tres instrucciones comprendidas entre ELSE y ENDIF. Esto no cambiaría el funcionamiento del programa. Sin embargo, sí se perdería la opción de ejecutar estas tres instrucciones si es que se realiza un cambio a la constante en la definición de VERSION.

4. IFDEF <alias>
   Esta directiva trabaja directamente con las directivas ELSE y ENDIF. La diferencia es que evalúa simplemente si un alias ha sido creado. De ser cierta la afirmación, se compilará el código comprendido entre IFDEF-ELSE o IFDEF-ENDIF en caso de que no se utilice la opción ELSE. En el programa 4.19 se muestra un ejemplo de uso de esta directiva.

Programa 4.19. Ejemplo de uso de la directiva IFDEF

```
#define SUMA_10 addlw .10      ;El texto SUMA_10 es un alias a la instrucción addlw  .10
#define VERSION  2             ;El texto VERSION es un alias a la constant 2
        org 0x00               ;Vector de Reset
        goto MAIN              ;Se salta a la dirección 0x200 de la memoria de programa.
        org 0x200
MAIN:
        IFDEF VERSION          ;El alias  VERSION ha sido creado por tanto la
        clrf 0x20,0            ;condición es verdadera. Entonces
        movlw .5               ;se compilan las instrucciones entre el IFDEF-ELSE.
        SUMA_10
        SUMA_10
        ELSE                   ;La condición falsa no se ejecuta y por tanto las
        setf 0x20,0            ;instrucciones comprendidas entre ELSE y ENDIF no se
        movlw .3               ;compilan.
        SUMA_10
        ENDIF
```

5. IFNDEF <alias>
   Esta directiva es muy similar a la IFDEF, con la diferencia de que la condición verdadera será cuando el alias no haya sido creado. En el programa 4.20 se muestra un ejemplo de uso de esta directiva.

Programa 4.20. Ejemplo de uso de la directiva IFNDEF <alias>

```
#define SUMA_10 addlw .10      ;El texto SUMA_10 es un alias a la instrucción addlw  .10
#define VERSION  2             ;El texto VERSION es un alias a la constant 2
        org 0x00               ;Vector de Reset
        goto MAIN              ;Se salta a la dirección 0x200 de la memoria de programa
        org 0x200
```

```
MAIN:
        IFNDEF VERSION      ;El alias  VERSION existe por lo tanto la condición es
                            ;falsa. Entonces se compilan las instrucciones entre
        clrf 0x20,0         ;ELSE – ENDIF.
        movlw .5
        SUMA_10
        SUMA_10
        ELSE                ;Esta condición es verdadera y por tanto se compilan las
        setf 0x20,0         ;instrucciones comprendidas entre ELSE y ENDIF.
        movlw .3
        SUMA_10
        ENDIF
```

6. WHILE <condición> – ENDW

   Las directivas WHILE – ENDW permiten definir un bloque de código que se va a repetir un conjunto de veces no mayor a 256. Para definir la cantidad de repeticiones, se debe utilizar otra directiva (que se verá más adelante) que permite crear una variable. La variable se inicializa en un valor y, mientras este valor se encuentre dentro de un rango, la condición será verdadera y el bloque se repetirá hasta que la condición sea falsa. En el programa 4.21 se muestra un ejemplo de estas directivas.

Programa 4.21. Ejemplo de uso de las directivas WHILE – ENDW

```
VARIABLE i          ;Se crea la variable i.
        org 0x00    ;Dirección del vector de Reset.
        goto MAIN   ;Se ejecuta la instrucción goto MAIN al inicio.
        org 0x200
MAIN:
        movlw .5    ;Se carga el registro W con el valor decimal .5
i=0                 ;La variable i se inicializa en 0.
        WHILE (i<5) ;Mientras i sea menor a 5 se repetirán las
                    ;instrucciones entre WHILE – END.
        addlw .2    ;W se suma con el valor decimal .2.
        nop         ;No se realiza ninguna operación.
i+=1                ;Se incrementa la variable i en 1.
        ENDW        ;Se vuelve a evaluar la condición.
                    ;Al salir del bucle WHILE, el valor de W será .15 ya
                    ;que se sumó .2 al valor inicial de.5, un número
                    ;total de 5 veces.
```

7. CODE

   Es una directiva que permite crear un nombre a un sector de código. Esto se utiliza en el archivo objeto para sectores de código con nombres identificadores. Por ejemplo, para definir un sector RESET, se utilizan las siguientes sentencias:

```
1. RESET code 0x01FF
2. goto INICIO
```

8. #INCLUDE<nombre del archivo>

   La directiva #include permite incluir un archivo con código o directivas importantes para que puedan ser usadas por el programa principal. Se pueden tener archivos con macros, rutinas de programas o directivas que declaran constantes. Cada archivo se puede incluir al proyecto indicando su nombre y extensión, de la siguiente manera:

   #include <P18F4550.INC>

   Cabe resaltar que esta directiva, por recomendación, debe ir al inicio del archivo de texto, para evitar problemas de invocación a alias o rutinas inexistentes. Se puede incluir la ruta del archivo también dentro de la directiva #INCLUDE, pero colocando el texto entre comillas dobles. Por ejemplo:

   #include "D:\\APOYO.INC"

9. RADIX

   Este operador hace referencia a la base por defecto que tienen las cifras en el programa. Todos los programas vistos hasta el momento tienen una base por defecto hexadecimal, a menos que se especifique lo contrario. Por ejemplo, se ejecuta la siguiente instrucción: movlw 14. La cifra 14 no tiene especificada una base, ya que se utiliza 0x14 si la cifra es hexadecimal, .14 si la cifra es decimal y 14O si la cifra se encuentra en base octal; pero en la instrucción mostrada el valor cargado en el registro W será 0x14, ya que la base por defecto es hexadecimal.

   Con el operador RADIX es posible modificar la base por defecto a decimal, octal o hexadecimal. En el programa 4.22 se muestra un ejemplo del cambio de base.

Programa 4.22. Cambio de base por defecto haciendo uso del operador RADIX

```
MAIN:
        radix dec       ;Se selecciona base decimal por defecto
        movlw 9         ;Se carga el valor decimal 9 en el registro W
        radix hex       ;Se selecciona base hexadecimal por defecto
        movlw 12        ;Se ingresa el valor 0x12 al registro W
        radix oct       ;Se selecciona base octal por defecto
        movlw 13        ;Se ingresa el valor 0x0B o .11 en el registro W
```

10. END

La directiva END determina el fin de un programa. Este operador es imprescindible, ya que, de obviarse, el compilador mostrará un error indicando que no ubica el final del programa. Cualquier instrucción por debajo del operador END será ignorada por el compilador.

El programador debe evitar que el PC llegue a la posición del END, ya que, de ocurrir el fin del programa, el CPU ingresará en estado de Reset, reiniciándose desde el vector de Reset.

Programa 4.23. Ejemplo de uso del operador END

```
MAIN:
        clrf 0x20,0          ;Se pone a 0 el registro de la dirección 0x20 de la SRAM.
        movlw .10            ;Se carga el registro W con el valor decimal .10.
        movwf 0x20,0         ;Se carga el registro de la dirección 0x20 de la SRAM con el
                             ;valor .10.
FIN:
        goto FIN             ;Se llama a la etiqueta FIN de forma indefinida. Esto evita
                             ;llegar al punto donde se Encuentra la directiva END.
END
        movlw .15            ;Esta instrucción es ignorada por el compilador ya que se
                             ;encuentra después de la directiva END.
```

Por otra parte, las directivas de datos se utilizan para definir bloques de memoria de programa con valores iniciales o para establecer alias a direcciones de la memoria de datos. Estas directivas son muy necesarias, ya que muchas de estas mantienen relación directa con las direcciones de memoria que el programa va a interactuar.

Gráfico 4.15. Principales directivas de datos del compilador MPASM

Directiva	Función
CBLOCK	Permite definir un bloque de alias a direcciones consecutivas de la memoria de datos.
ENDC	Finaliza la directiva CBLOCK.
CONSTANT	Permite crear un alias constante cuyo valor no puede ser modificado.
DA	Crea un arreglo de datos en memoria de programa en formato ASCII.
DATA	Permite crear un arreglo de datos en memoria de programa en diversas bases numéricas y haciendo uso de alias.
DB	Permite crear una tabla de bytes de datos, donde cada elemento es un byte.
DT	Genera una serie de instrucciones de retlw, donde cada byte se almacena en una posición par de la memoria de programa.
DW	Permite crear una tabla de palabras de 16 bits de datos.

Directiva	Función
EQU	Crea un alias de un valor numérico. Normalmente se utiliza para definir alias a direcciones de la memoria de datos.
FILL	Permite llenar la memoria de programa con un valor fijo hasta una cierta posición de la memoria de programa.
RES	Establece un espacio de memoria de programa de un tamaño específico.
SET	Esta directiva es equivalente a EQU, con la diferencia de que puede ser modificada más adelante en el programa.
VARIABLE	Es equivalente a la directiva SET, con la diferencia de que una variable no necesita tener un valor inicial y puede ser incrementada y decrementada durante el programa.

Elaboración propia.

1. CBLOCK <valor>– ENDC
   Estas directivas permiten crear un conjunto de alias a valores numéricos asignando estas cifras de forma ascendente conforme los alias son declarados. Inicialmente, el parámetro valor especifica la cifra inicial sobre la cual los alias estarán declarados. Por ejemplo, CBLOCK .6 indica que el primer alias tendrá asignado el valor .6, el segundo el valor .7, y así consecutivamente hasta que se llegue a la directiva ENDC, que marca el fin de la declaratoria de alias. En el programa 4.24 se muestra un ejemplo de uso de estas directivas.

Programa 4.24. Ejemplo de uso de las directivas CBLOCK y ENDC

```
        CBLOCK 0x10         ;Se define un bloque de alias, donde el primer valor asume
                            ;la constante 0x10.
            var1            ;var1 es un alias a 0x10.
            var2            ;var2 es un alias a 0x11.
            var3            ;var3 es un alias a 0x12.
        ENDC                ;Fin de la directiva CBLOCK.
        org 0x00            ;Vector de Reset.
        goto MAIN           ;Salto a la etiqueta MAIN (dirección 0x20 de la memoria de
                            ;programa).
        org 0x20            ;Dirección de inicio 0x20 de la memoria de programa.
MAIN:
        movlw .9            ;Se carga el valor de .9 en el registro W.
        movwf var1,0        ;Se carga el valor de W en la dirección absoluta 0x10 de la
                            ;SRAM
        incf  var2,f,0      ;Se incrementa el contenido del registro 0x11 en uno
        movff var2,var3     ;Se carga el contenido del registro 0x11 a la dirección 0x12
FIN:
        goto FIN            ;Bucle infinito
END
```

2. CONSTANT <alias> <constante>
   Esta directiva permite crear un alias a un valor constante o literal que no puede ser modificado más adelante en el programa. En el programa 4.25 se muestra un ejemplo de funcionamiento de esta directiva.

Programa 4.25. Ejemplo de uso de la directiva CONSTANT

```
        CBLOCK 0x10
              var1                ;Se crea un alias var1 equivalente al valor 0x10
        ENDC
        CONSTANT TIEMPO =.50      ;Se crea una constante TIEMPO que es un alias a
                                  ;.50
        org 0x00                  ;Vector de Reset.
        goto MAIN                 ;Salto a la etiqueta MAIN (dirección 0x20 de la
                                  ;memoria de programa)
        org 0x20                  ;Dirección de inicio 0x20 de la memoria de programa.
MAIN:
        clrf TRISD                ;Todos los pines del Puerto D se configuran como
                                  ;salida digital.
        clrf LATD                 ;Se ponen todos los pines del Puerto D a estado '0'
                                  ;lógico.
INICIO:
        call RETARDO              ;Se invoca a la rutina RETARDO.
        incf LATD,f               ;Se incrementa el Puerto D en un dígito binario.
        goto INICIO               ;Se retorna a la etiqueta INICIO.
RETARDO:
        movlw TIEMPO              ;Se carga el valor TIEMPO (.50) en el registro W.
        movwf var1                ;Se aloja el valor .50 en la dirección 0x10.
RET1:
        decfsz var1,f             ;Se resta el contenido de var1 en 1 y se compara
                                  ;con 0
        goto   RET1               ;Si es diferente de 0 se retorna a RET1
        return                    ;Si es igual a 0 se ejecuta el return y se ejecuta la
                                  ;instrucción
END                               ;después de call RETARDO
```

3. DA <arreglo de bytes ASCII>
   Esta directiva permite crear una secuencia de bytes en formato ASCII dentro de la memoria de programa. Primero se tiene que definir la dirección de inicio de la memoria de programa donde se creará el arreglo de datos. Es recomendable utilizar una dirección alejada de la zona donde estarán las instrucciones del programa. Además, es altamente útil el uso de una etiqueta para tener un alias a la dirección de inicio del arreglo de datos, ya que con el registro puntero de tabla (TBLPTR) es posible acceder a todos los elementos del arreglo. El último elemento del arreglo será un byte en

código ASCII seguido de un byte 0xFF, que corresponde a un valor fuera del arreglo, lo cual indica el fin de la tabla de datos.

En el programa 4.26 se muestra un ejemplo de la directiva DA. Se crea un arreglo de datos en la dirección 0x20 con el mensaje ASCII "Probando la programación en el PIC18F" compuesto de 36 bytes. Como se almacenan 2 bytes por cada dirección, se tienen en total 18 direcciones de la memoria de programa, lo cual hace que el arreglo ocupe el rango de direcciones de 0x20-0x32. En la memoria de datos se almacenarán los 36 bytes del arreglo, comenzando desde la dirección 0x10 hasta la 0x44. Cabe resaltar que con la directiva DA no se desperdician espacios de memoria de programa, se aprovechan los 16 bits entregados por cada dirección par.

Programa 4.26. Ejemplo de uso de la directiva DA

```
        org 0x00                ;Vector de Reset.
        goto MAIN               ;Salto a la etiqueta MAIN.
        org 0x20                ;Dirección de inicio 0x20 de la memoria de programa.
TABLA: da "Probando la programacion en el PIC18F"    ;Se crea el arreglo donde
                                                     ;TABLA es una etiqueta que
        org 0x200               ;apunta a la dirección 0x20. Se inicia el programa en la
                                ;dirección 0x200.
MAIN:
        movlw UPPER TABLA       ;Se coloca en el registro W el byte alto de la dirección
                                ;TABLA.
        movwf TBLPTRU           ;Se carga TBLPTRU con el valor 0x00.
        movlw HIGH TABLA        ;Se carga el registro W con el byte alto de la
                                ;dirección TABLA.
        movwf TBLPTRH           ;Se carga TBLPTRH con el valor 0x00.
        movlw LOW  TABLA        ;Se carga el registro W con el byte bajo de la dirección
                                ;TABLA.
        movwf TBLPTRL           ;Se carga el registro TBLPTRL con el valor 0x20.
        lfsr  FSR0,0x010        ;Se apunta a la dirección SRAM 0x10 con el registro
                                ;FSR0.
INICIO:
        TBLRD*+                 ;Se lee un elemento del arreglo  TABLA y se
                                ;incrementa el puntero TBLPTR.
        movf  TABLAT,W          ;Se carga el contenido apuntado por TBLPTR al
                                ;registro W.
        movwf 0x01,0            ;Se carga el valor del registro W a la dirección absoluta
                                ;0x01 de la SRAM.
        movlw 0xFF              ;Se carga el valor 0xFF en el registro W.
        xorwf 0x01,W            ;Se compara 0xFF con el valor leído de la tabla.
        btfsc STATUS,Z          ;Si son iguales Z=1, entonces se salta a FIN. En caso
                                ;contrario.
        goto  FIN               ;Salto a la etiqueta FIN si se llega al final de la lectura
                                ;de TABLA.
```

```
        movf  0x01,W        ;Z=0, el programa continua. Se lee el valor de
                            ;TABLA almacenado en la
        movwf INDF0         ;dirección absoluta 0x01 y se carga en INDF0.
        incf  FSR0L,f       ;Se incrementa el puntero de direcciones de la
                            ;memoria SRAM.
        goto  INICIO        ;Se salta a la etiqueta INICIO
FIN:
        goto FIN            ;Fin del programa, bucle infinito
END
```

4. DATA <arreglo de bytes>

   Esta directiva es muy similar a DA, con la diferencia de que el arreglo de datos puede comprender valores en cualquier formato, ya sea ASCII, decimal, hexadecimal, binario, octal, valores de 8 y 16 bits, e inclusive soporta alias creados por otras directivas. En este arreglo también se aprovechan al máximo las direcciones de la memoria de programa, llegando a almacenarse 2 bytes por cada dirección par. En el programa 4.27 se muestra un ejemplo de uso de esta directiva. El programa realiza la misma función que el programa 4.26, con la diferencia de que el arreglo de datos en TABLA contiene otros valores.

Programa 4.27. Ejemplo de uso de la directiva DATA

```
        VAR EQU .2                  ;Se crea el alias VAR equivalente al valor de 2.
        org 0x00                    ;Vector de Reset.
        goto MAIN                   ;Se salta a la etiqueta MAIN (dirección 0x200).
        org 0x20                    ;Se define la dirección 0x20 de la memoria de programa.
TABLA: data "PIC18F",VAR,'N',.128   ;Se crea el arreglo de datos, donde la
                                    ;ETIQUETA TABLA apunta a la primera
        org 0x200                   ;dirección del arreglo. Se inicia el programa en la dirección
                                    ;0x20.
MAIN:
        movlw UPPER TABLA           ;Se coloca en el registro W el byte alto de la dirección
                                    ;TABLA
        movwf TBLPTRU               ;Se carga TBLPTRU con el valor 0x00
        movlw HIGH TABLA            ;Se carga el registro W con el byte alto de la dirección
                                    ;TABLA
        movwf TBLPTRH               ;Se carga TBLPTRH con el valor 0x00
        movlw LOW  TABLA            ;Se carga el registro W con el byte bajo de la dirección
                                    ;TABLA.
        movwf TBLPTRL               ;Se carga el registro TBLPTRL con el valor 0x20.
        lfsr  FSR0,0x010            ;Se apunta a la dirección SRAM 0x10 con el registro FSR0.
```

```
INICIO:
        TBLRD*+          ;Se lee un elemento del arreglo  TABLA y se incrementa el
                         ;puntero TBLPTR.
        movf  TABLAT,W   ;Se carga el contenido apuntado por TBLPTR al registro W
        movwf 0x01,0     ;Se carga el valor del registro W a la dirección absoluta 0x01
                         ;de la SRAM.
        movlw 0xFF       ;Se carga el valor 0xFF en el registro W.
        xorwf 0x01,W     ;Se compara 0xFF con el valor leído de la tabla.
        btfsc STATUS,Z   ;Si son iguales Z=1, entonces se salta a FIN. En caso
                         ;contrario.
        goto  FIN        ;Salto a la etiqueta FIN si se llega al final de la lectura de
                         ;TABLA.
        movf  0x01,W     ;Z=0, el programa continua. Se lee el valor de TABLA
                         ;almacenado en la dirección absoluta 0x01 y se carga en
        movwf INDF0      ;INDF0.
        incf  FSR0L,f    ;Se incrementa el puntero de direcciones de la memoria
                         ;SRAM.
        goto  INICIO     ;Se salta a la etiqueta INICIO
FIN:
        goto FIN         ;Fin del programa, bucle indefinido
        END
```

Cabe mencionar que en el arreglo declarado en TABLA, cada coma permite concatenar un nuevo elemento con el anterior, pero al mismo tiempo implica una dirección diferente. Como se puede observar, el arreglo se encuentra compuesto por la cadena "PIC18F", la cual contiene 6 bytes que ocupan tres direcciones de memoria de programa (0x20-0x22). Posteriormente, el alias VAR, que equivale a 0x02, ocupará las direcciones 0x23 y 0x24 con el valor 0x00 y 0x02, respectivamente (se coloca un byte 0x00 para completar los 16 bits de la dirección par de memoria). Como resultado del programa, se tendrá la siguiente secuencia en las direcciones 0x10, lo mismo ocurrirá para el byte ASCII 'N' y para el valor decimal .128. Por lo tanto, las direcciones de la memoria de datos estarán configuradas de la siguiente manera, como se observa en el gráfico 4.16.

Gráfico 4.16. Valores de la tabla de datos alojados en la memoria SRAM como resultado del programa 4.27

Dirección de la memoria de datos	Valor de la tabla de datos
0x10	'P'
0x11	'I'
0x12	'C'
0x13	'1'
0x14	'8'
0x15	'F'

Dirección de la memoria de datos	Valor de la tabla de datos
0x16	0x02
0x17	0x00
0x18	'N'
0x19	0x00
0x1A	0x80
0x1B	0x00

Elaboración propia.

5. DB <arreglo de bytes>
   La directiva DB permite definir los caracteres de 8 bits del arreglo de datos uno por uno separados por una coma. Los parámetros pueden estar en cualquier base, siempre y cuando sean elementos de un byte de longitud. En el ejemplo del programa 4.28 se realiza la misma aplicación que en los dos programas anteriores. La diferencia se encuentra en la declaración de la etiqueta TABLA que contiene un mensaje de 7 bytes, cada elemento separado por una coma. El compilador entiende que este mensaje deberá ocupar 4 direcciones de la memoria de programa. La última dirección que corresponde al quinto byte se le agregará un byte 0x00 por delante.

Programa 4.28. Ejemplo de uso de la directiva DB

```
org 0x00                        ;Vector de Reset
        goto MAIN               ;Se salta a la etiqueta MAIN (dirección 0x200)
        org 0x20                ;Se define la dirección 0x20 de la memoria de programa.
TABLA: db 'P','I','C',0x31,0x38,'F',.5     ;Se crea un arreglo de bytes de 7 elementos.
        org 0x200
MAIN:
        movlw UPPER TABLA       ;Se coloca en el registro W el byte alto de la
                                ;dirección TABLA.
        movwf TBLPTRU           ;Se carga TBLPTRU con el valor 0x00
        movlw HIGH TABLA        ;Se carga el registro W con el byte alto de la
                                ;dirección TABLA.
        movwf TBLPTRH           ;Se carga TBLPTRH con el valor 0x00
        movlw LOW  TABLA        ;Se carga el registro W con el byte bajo de la
                                ;dirección TABLA.
        movwf TBLPTRL           ;Se carga el registro TBLPTRL con el valor 0x20
        lfsr  FSR0,0x010        ;Se apunta a la dirección SRAM 0x10 con el
                                ;registro FSR0.
INICIO:
        TBLRD*+                 ;Se lee un elemento del arreglo  TABLA y se
                                ;incrementa el puntero TBLPTR.
        movf  TABLAT,W          ;Se carga el contenido apuntado por TBLPTR al
                                ;registro W
```

```
        movwf 0x01,0        ;Se carga el valor del registro W a la dirección
                            ;absoluta 0x01 de la SRAM.
        movlw 0xFF          ;Se carga el valor 0xFF en el registro W
        xorwf 0x01,W        ;Se compara 0xFF con el valor leído de la tabla
        btfsc STATUS,Z      ;Si son iguales Z=1, entonces se salta a FIN. En
                            ;caso contrario.
        goto  FIN           ;Salto a la etiqueta FIN si se llega al final de la
                            ;lectura de TABLA.
        movf  0x01,W        ;Z=0, el programa continua. Se lee el valor de
                            ;TABLA almacenado en la
        movwf INDF0         ;dirección absoluta 0x01 y se carga en INDF0.
        incf  FSR0L,f       ;Se incrementa el puntero de direcciones de la
                            ;memoria SRAM.
        goto  INICIO        ;Se salta a la etiqueta INICIO
FIN:
        goto FIN            ;Fin del programa, bucle infinito
        END
```

En el gráfico 4.17 se muestra el resultado de cargar en memoria de datos los bytes del arreglo definido por la directiva DB. Como se puede observar, se aprovechan al máximo las direcciones de la memoria de programa.

Gráfico 4.17. Valores de la tabla de datos alojados en la memoria SRAM como resultado del programa 4.27

Dirección de la memoria de datos	Valor de la tabla de datos
0x10	'P'
0x11	'I'
0x12	'C'
0x13	0x31
0x14	0x38
0x15	'F'
0x16	0x05
0x17	0x00

Elaboración propia.

6. DT <arreglo de bytes>
   La directiva DT permite generar un arreglo de bytes similar a DB, la diferencia es la equivalencia con la instrucción retlw, donde cada byte ocupará una instrucción par, desperdiciándose una dirección impar de la memoria de programa. En el programa 4.29 se puede ver un ejemplo de uso de esta directiva.

Programa 4.29. Ejemplo de uso de la directiva DT

```
        org 0x00              ;Vector de Reset
        goto MAIN             ;Salto a la etiqueta MAIN
        org 0x20              ;Dirección de inicio 0x20 de la memoria de programa
TABLA: DT "Hola",'P',.10      ;Se define el arreglo de un total de 6 caracteres
                              ;equivalentes a 12 bytes
        org 0x200
MAIN:
        movlw UPPER TABLA     ;Se coloca en el registro W el byte alto de la
                              ;dirección TABLA
        movwf TBLPTRU         ;Se carga TBLPTRU con el valor 0x00
        movlw HIGH TABLA      ;Se carga el registro W con el byte alto de la
                              ;dirección TABLA.
        movwf TBLPTRH         ;Se carga TBLPTRH con el valor 0x00
        movlw LOW  TABLA      ;Se carga el registro W con el byte bajo de la
                              ;dirección TABLA.
        movwf TBLPTRL         ;Se carga el registro TBLPTRL con el valor 0x20
        lfsr  FSR0,0x010      ;Se apunta a la dirección SRAM 0x10 con el registro
                              ;FSR0.
INICIO:
        TBLRD*+               ;Se lee un elemento del arreglo  TABLA y se
                              ;incrementa el puntero TBLPTR.
        movf  TABLAT,W        ;Se carga el contenido apuntado por TBLPTR al
                              ;registro W.
        movwf 0x01,0          ;Se carga el valor del registro W a la dirección
                              ;absoluta 0x01 de la SRAM.
        movlw 0xFF            ;Se carga el valor 0xFF en el registro W.
        xorwf 0x01,W          ;Se compara 0xFF con el valor leído de la tabla
        btfsc STATUS,Z        ;Si son iguales Z=1, entonces se salta a FIN. En
                              ;caso contrario.
        goto  FIN             ;Salto a la etiqueta FIN si se llega al final
                              ;de la lectura de TABLA.
        movf  0x01,W          ;Z=0, el programa continua. Se lee el valor de
                              ;TABLA almacenado en la dirección absoluta 0x01
        movwf INDF0           ;y se carga en INDF0.
        incf  FSR0L,f         ;Se incrementa el puntero de direcciones de la
                              ;memoria SRAM.
        goto  INICIO          ;Se salta a la etiqueta INICIO.
FIN:
        goto FIN              ;Fin del programa, bucle indefinido.
        END
```

Al enviar los datos de la tabla a la memoria de SRAM, se podrá ver que se desperdicia un byte con dirección impar con el valor 0x0C. Esto se observa como resultado del programa en el gráfico 4.18.

Gráfico 4.18. Valores de la tabla de datos alojados en la memoria SRAM como resultado del programa 4.29

Dirección de la memoria de datos	Valor de la tabla de datos
0x10	'H'
0x11	0x0C
0x12	'o'
0x13	0x0C
0x14	'l'
0x15	0x0C
0x16	'a'
0x17	0x0C
0x18	'P'
0x19	0x0C
0x1A	0x0A
0x1B	0x0C

Elaboración propia.

7. DW <arreglo de words>

   La directiva DW (*Define Word*) permite establecer un arreglo de datos de cifras de 16 bits de longitud, separados por comas. Con esto se aprovechan al máximo las direcciones de la memoria de programa.

Programa 4.30. Ejemplo de uso de la directiva DW

```
        org 0x00                ;Vector de Reset.
        goto MAIN               ;Salto a la etiqueta MAIN.
        org 0x20                ;Dirección de inicio 0x20 de la memoria de programa.
TABLA: DW .32768,0x1234,0x456,0x45 ;Se define un arreglo de cuatro elementos
                                   ;de 16 bits cada uno
        org 0x200
MAIN:
        movlw UPPER TABLA       ;Se coloca en el registro W el byte alto de la
                                ;dirección TABLA.
        movwf TBLPTRU           ;Se carga TBLPTRU con el valor 0x00.
        movlw HIGH TABLA        ;Se carga el registro W con el byte alto de la
                                ;dirección TABLA.
        movwf TBLPTRH           ;Se carga TBLPTRH con el valor 0x00.
```

```
        movlw LOW  TABLA        ;Se carga el registro W con el byte bajo de la
                                ;dirección TABLA.
        lfsr  FSR0,0x010        ;Se apunta a la dirección SRAM 0x10 con el
                                ;registro FSR0.
        movwf TBLPTRL           ;Se carga el registro TBLPTRL con el valor
                                ;0x20.
INICIO:
        TBLRD*+                 ;Se lee un elemento del arreglo  TABLA y
                                ;se incrementa el puntero TBLPTR.
        movf  TABLAT,W          ;Se carga el contenido apuntado por
                                ;TBLPTR al registro W.
        movwf 0x01,0            ;Se carga el valor del registro W a la
                                ;dirección absoluta 0x01 de la SRAM.
        movlw 0xFF              ;Se carga el valor 0xFF en el registro W.
        xorwf 0x01,W            ;Se compara 0xFF con el valor leído de la
                                ;tabla.
        btfsc STATUS,Z          ;Si son iguales Z=1, entonces se salta a FIN.
                                ;En caso contrario
        goto  FIN               ;Salto a la etiqueta FIN si se llega al final de
                                ;la lectura de TABLA.
        movf  0x01,W            ;Z=0, el programa continua. Se lee el valor
                                ;de TABLA almacenado en la dirección
        movwf INDF0             ;absoluta 0x01 y se carga en INDF0.
        incf  FSR0L,f           ;Se incrementa el puntero de direcciones
                                ;de la memoria SRAM.
        goto  INICIO            ;Se salta a la etiqueta INICIO.
FIN:
        goto FIN                ;Fin del programa, bucle indefinido.
        END
```

Como resultado del programa 4.30, se tienen los siguientes bytes del arreglo de datos ordenados en la memoria SRAM de la manera como se observa en el gráfico 4.19.

Gráfico 4.19. Valores de la tabla de datos alojados en la memoria SRAM como resultado del programa 4.30

Dirección de la memoria de datos	Valor de la tabla de datos
0x10	0x00
0x11	0x80
0x12	0x34
0x13	0x12
0x14	0x56
0x15	0x04
0x16	0x45
0x17	0x00

Elaboración propia.

8. EQU <alias> <constante>
   Esta directiva permite crear un alias a un valor constante. Se utiliza normalmente para definir direcciones de memoria de datos y hacer más legible el programa, ya que a una dirección se le puede asociar un alias con su función. En el programa 4.31 se muestra un ejemplo de esta directiva.

Programa 4.31. Ejemplo de uso de la directiva EQU

```
AUX EQU 0x20          ;Se crea un alias de nombre AUX a 0x20.
        org 0x00      ;Vector de Reset.
        goto MAIN     ;Salto a la etiqueta MAIN.
        org 0x200     ;Dirección de inicio 0x200 de la memoria de programa.
MAIN:
        movlw .45     ;Se carga el registro W con el valor decimal .45.
        movwf AUX,0   ;Se envía el valor decimal .45 a la dirección absoluta 0x20.
FIN:
        goto FIN      ;Bucle indefinido.
        END
```

9. FILL <valor> <tamaño en direcciones de llenado>
   La directiva FILL permite inicializar un sector de la memoria de programa con un valor fijo. Por ejemplo, en el programa 4.32 se cargan las direcciones 0x20 a 0x11E con el valor 0x00 y se muestra un ejemplo de esta directiva.

Programa 4.32. Ejemplo de uso de la directiva FILL

```
AUX EQU 0x20              ;Se crea un alias de nombre AUX a 0x20
        org 0x00          ;Vector de Reset
        goto MAIN         ;Salto a la etiqueta MAIN
        org 0x20          ;Dirección de inicio 0x20 del llenado de datos.
        FILL 0x00,0x100   ;Cargar las direcciones 0x020-0x11E de la memoria de
                          ;programa con 0x00.
        org 0x200
MAIN:
        movlw .45         ;Se carga el registro W con el valor decimal .45.
        movwf AUX,0       ;Se envía el valor decimal .45 a la dirección absoluta 0x20.
FIN:
        goto FIN          ;Bucle indefinido.
        END
```

10. <nombre del sector > RES <tamaño>

    La directiva RES permite reservar un espacio de la memoria de programa. Este espacio se genera desde el valor especificado por la directiva org y el tamaño dado en RES. En el programa 4.33 se muestra un ejemplo. La directiva NOMBRE RES 0x10 reserva 16 espacios de memoria a partir de la dirección 0x20. Es decir, se reservan los espacios 0x20-0x2E. Luego, se llenan 0x20 espacios (NOMBRE es alias de 0x20) con 0x00, logrando inicializar el valor 0x00 entre las direcciones 0x30 y 0x4E (0x20 o 32 espacios en total).

Programa 4.33. Ejemplo de uso de la directiva RES

```
        org 0x00          ;Vector de Reset
        goto MAIN         ;Salto a la etiqueta MAIN
        org 0x20          ;Dirección de inicio 0x20 del llenada de datos en la
                          ;memoria de programa
        NOMBRE RES 0x10   ;Se reserva un espacio de 16 direcciones de memoria
                          ;con NOMBRE desde la dirección 0x20 hasta la 0x30.
        FILL 0x00,NOMBRE  ;Se con llena con 0x00 las 16 direcciones de la 0x30
        org 0x200         ;a la 0x40.
MAIN:
        movlw 0x90        ;Se carga el valor 0x90 en el registro W.
FIN
        goto FIN          ;Bucle indefinido.
        END
```

11. <alias> SET valor

    La directiva SET es muy similar a la directiva EQU, con la ventaja de que con esta primera es posible redeclarar valores de alias previamente creados. Por ejemplo, en el programa 4.34 se asignan dos valores diferentes al mismo alias VALOR.

Programa 4.34. Ejemplo de uso de la directiva SET

```
VALOR SET 0x10          ;Se define el alias VALOR equivalente a 0x10.
        org 0x00        ;Vector de Reset.
        goto MAIN       ;Salto a la etiqueta MAIN.
        org 0x200       ;Dirección 0x200 de inicio del programa.
MAIN:
        movlw VALOR     ;Se carga en el registro W el valor 0x10.
        movwf 0x20,0    ;Se guarda en la dirección 0x20 de la SRAM el valor 0x10.
VALOR SET 0x4           ;Se redefine el alias VALOR como 0x04.
        movlw VALOR     ;Se carga en el registro W el valor 0x04.
FIN
        goto FIN        ;Bucle indefinido.
        END
```

12. VARIABLE <alias> = <valor inicial>

    La directiva VARIABLE es similar al SET, con la diferencia de que, aquí, este alias se puede incrementar o restar de acuerdo con una operación aritmética que se coloca en un punto del programa. De esta manera, la variable puede interactuar con los registros del microcontrolador, asignándole diversos valores.

Programa 4.35. Ejemplo de uso de la directiva VARIABLE

```
VARIABLE num = .4       ;Se crea una variable de nombre num que es equivalente al
                        ;valor .4.
        org 0x00        ;Vector de Reset.
        goto MAIN       ;Salto a la etiqueta MAIN.
        org 0x200       ;Dirección 0x200 de inicio del programa.
MAIN:
        movlw num       ;Se carga el registro W con el valor .4.
        movwf 0x20,0    ;Se envía el valor .4 a la dirección absoluta 0x20 de la
                        ;memoria SRAM.
        num = num + .2  ;Se incrementa num en .2, es decir, ahora vale .6.
        movlw num       ;Se carga el registro W con el valor .6.
        movwf 0x21,0    ;Se envía el valor .6 a la dirección absoluta 0x21 de la
                        ;memoria SRAM.
FIN:
        goto FIN        ;Bucle indefinido.
END
```

Las directivas informativas permiten configurar las opciones de mensajes presentados en la ventana de salida del compilador MPASM del MPLAB X IDE. Algunas de las directivas informativas más conocidas se muestran en el gráfico 4.20.

Gráfico 4.20. Directivas informativas más utilizadas

Directiva	Función
ERROR	Permite generar un mensaje de error en la ventana de salida del compilador de MPLAB IDE.
ERRORLEVEL	Permite habilitar la presentación de errores, advertencias y mensajes.
MESSG	Permite mostrar un mensaje en la ventana de salida del compilador de MPLAB IDE.
SPACE	Genera espacios en blanco.

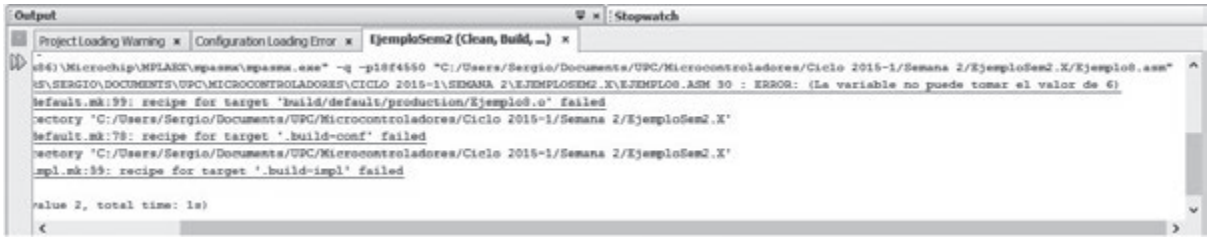
Elaboración propia.

1. ERROR <mensaje>
   La directiva ERROR permite editar un mensaje de error en el compilador, a fin de advertir determinada situación que debe impedir la correcta compilación del programa. El mensaje se coloca entre comillas dobles luego de la directiva ERROR. En el programa 4.36 se muestra un ejemplo, en el cual se desplegará el mensaje de error en la ventana de salida del compilador si la variable num toma el valor de .6. Dado que al analizar el programa se ve que esta condición es cierta, el mensaje de error aparece tal como se muestra en el gráfico 4.21.

Programa 4.36. Ejemplo de uso de la directiva ERROR

```
VARIABLE num = .4
        org 0x00
        goto MAIN
        org 0x200
MAIN:
        movlw num
        movwf 0x20,0
num = num + .2;
        movlw num
        movwf 0x21,0
 if num == 6                ;directiva error que genera el mensaje entre comillas
        error "La variable no puede tomar el valor de 6"
 endif                                                  ;dobles en el compilador
FIN:
        goto FIN
        END
```

Gráfico 4.21. Error generado por la directiva ERROR.
Se observa el mensaje generado en la ventana de salida



Elaboración propia.

2. ERRORLEVEL <num>

   La directiva ERRORLEVEL permite configurar las opciones de mensajes de la ventana de salida del compilador. De acuerdo con el valor asignado en num, se tienen las siguientes configuraciones:

   1. num = 0 se pueden imprimir mensajes, advertencias y errores.
   2. num = 1 se pueden imprimir solo advertencias y errores.
   3. num = 2 solo se pueden imprimir errores.

   En el programa 4.37 se muestra un ejemplo de uso de esta directiva. Antes de mostrarse el mensaje de error aparece la directiva ERRORLEVEL 1, que habilita todos los mensajes a mostrar en la ventana de salida del compilador.

Programa 4.37. Ejemplo de uso de la directiva ERRORLEVEL

```
VARIABLE num = .4
        org 0x00
        goto MAIN
        org 0x200
MAIN:
        movlw num
        movwf 0x20,0
                num = num + .2;
        movlw num
        movwf 0x21,0
if num == 6
        ERRORLEVEL 1 ;Se habilitan los mensajes, advertencias y errores en la
                        ;ventana de salida del compilador
        error "La variable no puede tomar el valor de 6"
endif
FIN:
        goto FIN
        END
```

3. MESSG <mensaje>
   Esta directiva simplemente muestra un mensaje en la ventana de salida del compilador. El mensaje debe ir entre comillas dobles luego de la directiva MESSG. Por ejemplo, MESSG "Esta variable esta fuera de rango", muestra el mensaje entre comillas en la ventana de salida del compilador en la categoría de mensaje.

4. SPACE <num>
   Esta directiva simplemente muestra un número (num) de líneas en blanco en la ventana de salida del compilador.
   Finalmente, las directivas macro se utilizan para generar subprogramas que pueden luego ser invocados desde otro programa general o más grande. Las macros soportan argumentos, con lo cual el subprograma puede interactuar con valores iniciales de entrada, para ser más versátil y útil. En el gráfico 4.22 se muestran las principales directivas asociadas a las operaciones con macros.

Gráfico 4.22. Directivas macro

Directiva	Función
MACRO	Permite crear un subprograma que se puede invocar ingresando su nombre y argumentos.
ENDM	Define el fin de la macro o subprograma.
EXITM	Permite salir de una macro antes de que esta culmine.
EXPAND	Permite expandir la macro en el archivo lst.
NOEXPAND	No permite la expansión de la macro en el archivo lst.
LOCAL	

Elaboración propia.

1. <nombre> MACRO <argumento1, argumento2,…, argumento N> - ENDM
   La directiva MACRO permite definir un subprograma que será invocado con la etiqueta que se coloque por nombre. Posteriormente, la macro puede tener uno o más argumentos, que son parámetros que se ingresan al subprograma. Dentro de la macro existen instrucciones que ejecutan operaciones e interactúan con los argumentos. Finalmente, la macro acaba cuando se detecta la directiva ENDM. En el programa 4.38 se muestra un ejemplo de una macro que suma cuatro cifras literales y el resultado lo retorna en el registro de trabajo W.

Programa 4.38. Ejemplo de uso de las directivas MACRO y ENDM

```
SUMA_4 MACRO val1,val2,val3,val4     ;Definición de una macro de nombre
                                     ;SUMA_4 y cuatro argumentos
        movlw val1                   ;Instrucciones de la macro. Esta macro suma
                                     ;los valores literales
        addlw val2                   ;val1 + val2 + val3 + val4 y el resultado lo
                                     ;almacena en el registro W.
        addlw val3
        addlw val4
        ENDM                         ;Fin de la macro
        org 0x00                     ;Vector de Reset
        goto MAIN                    ;Se salta a la dirección 0x200
        org 0x200                    ;Inicio del programa en la dirección 0x200
MAIN:
        SUMA_4 0x01,0x02,0x03,0x04   ;Se invoca a la macro SUMA_4 y se pasan
                                     ;como parámetros 1,2,3 y 4
        movwf 0x20,0                 ;La macro retorna la suma .10 en W y se
                                     ;envía a la dirección 0x20 de la SRAM.
FIN
        goto FIN                     ;Bucle indefinido
END
```

2. EXITM

   Es una directiva similar a ENDM, pero no necesariamente debe ir al final de la macro, sino que puede estar en cualquier posición dentro. ENDM determina el fin de la macro antes de que llegue a su final. Por ejemplo, en el programa 4.39 se modifica la macro mostrada en el programa 4.38. En esta nueva macro, si el cuarto argumento es mayor que el valor decimal .240, se sale del proceso y no se ejecuta la sumatoria. Al invocar a la macro con los parámetros .1, .2, .3 y .245, el cuarto argumento resulta mayor que .240, con lo cual el valor del registro W no cambia.

Programa 4.39. Ejemplo de uso de la directiva EXITM

```
SUMA_4 MACRO val1,val2,val3,val4     ;Definición de la macro de nombre
                                     ;SUMA_4 y cuatro argumentos.
        if(val4>.240)                ;Si el cuarto argumento es mayor a .240 se
                                     ;termina la macro.
                EXITM                ;Directiva para salir de la macro.
        endif                        ;Fin del if.
        movlw val1                   ;val1 + val2 + val3 + val4 y el resultado se
                                     ;almacena en el registro W.
```

```
        addlw val2
        addlw val3
        addlw val4
        ENDM                        ;Fin de la macro

        org 0x00;Vector de Reset
        goto MAIN                   ;Se salta a la dirección 0x200.
        org 0x200                   ;Inicio del programa en la dirección 0x200.
MAIN:
        SUMA_4 .1,.2,.3,.245        ;Se invoca a la macro con el cuarto
                                    ;argumento mayor a .240.
        movwf 0x20,0                ;El valor de W se mantiene en su valor
                                    ;anterior 0x00 y este valor se envía.
FIN                                 ;a la dirección absoluta 0x20 de la memoria
                                    ;SRAM.
        goto FIN                    ;Bucle indefinido
        END
```

3. EXPAND – NOEXPAND
   Estas directivas se encargan de expandir (EXPAND) y no expandir (NOEXPAND) las macros dentro del archivo de listado .lst. Por expansión se entiende poder ver las instrucciones que contienen la macro.

4. LOCAL
   La directiva LOCAL permite redefinir algún alias creado de forma global. El nuevo alias asignado solo mantiene su valor en el contexto de la macro, luego, fuera de la macro, mantiene el valor dado de forma global. En el programa 4.40 se muestra un ejemplo de este tipo de macro.

Programa 4.40. Ejemplo de uso de la directiva LOCAL

```
VAR EQU .5                          ;Se crea el alias VAR que es equivalente al
                                    ;valor decimal .5.
SUMA_4 MACRO val1,val2,val3,val4    ;Se define la macro de nombre SUMA_4 y
                                    ;cuatro argumentos.
        LOCAL VAR                   ;Se indica que se va a redefinir el valor de
                                    ;VAR solo para el ámbito de la macro.
        VAR SET .3                  ;Se le asigna a VAR el valor de .3
        movlw val1                  ;La macro suma los valores literales val1 +
                                    ;val2+val3+ val4 + VAR (.3) y el resultado
        addlw val2                  ;se almacena en el registro W.
        addlw val3
```

```
        addlw val4
        addlw VAR
        ENDM
        org 0x00            ;Vector de Reset
        goto MAIN           ;Salto a la etiqueta MAIN
        org 0x200           ;Dirección de inicio en 0x200
MAIN:
        SUMA_4 .1,.2,.3,.4  ;Se invoca a la macro y se pasa como .1,.2,.3,.4
                            ;y argumentos se obtiene como resultado
        movwf VAR,0         ;1+2+3+4+3 = 13. El resultado se guarda en
                            ;W y se copia a la dirección VAR (.5) de la
FIN                         ;memoria SRAM.
        goto FIN            ;Bucle indefinido.
        END
```

Preguntas de repaso

1. ¿Qué diferencia hace si al final de una etiqueta se colocan los dos puntos (:)?
2. En el siguiente código en lenguaje ensamblador:

```
INICIO:         movwf  0x30,0,1
```

¿Cuál es el mnemónico?

3. En la siguiente operación en lenguaje ensamblador:

```
movlw 0x30
movwf 0x40,0
xorwf  0x40,f,0
```

¿Qué bits del registro STATUS quedan afectados?

4. Analice las siguientes directivas:

```
org 0x200
TABLA: da “Hola Mundo”
```

¿En qué dirección de la memoria de programa se encuentra almacenado el código ASCII ‘M’?

5. ¿Cuál es la relación entre los registros FSR0H, FSR0L e INDF0 en el modo de direccionamiento indirecto?
6. Explique cuál es la diferencia entre una dirección absoluta y una dirección relativa en la memoria de datos del PIC18F.
7. Analice el siguiente programa, que es ejecutado con un cristal externo de 8 MHz:

```
        var EQU 0x20
        org 0x00
        goto MAIN
        org 0x200
MAIN:
        movlw 0x10
        movwf var
LAZO:
        nop
        nop
        nop
        nop
        nop
        decfszvar,f
        goto LAZO
FIN:
        goto FIN
        END
```

¿Cuánto es el tiempo total de duración de la rutina LAZO?

8. Analice el siguiente programa:

```
var EQU 0x10
        org 0x00
        goto MAIN
        org 0x200
MAIN:
        clrf TRISD
INICIO:
        movlw .12
        movwf var
        incf  var
        call  ES_PAR_O_IMPAR
        nop
```

```
        goto FIN
ES_PAR_O_IMPAR:
        btfss var,0
        goto  PAR
        goto  IMPAR
PAR:
        movlw 0x0F
        movwf LATD
        return
IMPAR:
        movlw 0x01
        movwf LATD
        return
FIN:
        goto FIN
        END
```

¿Cuánto es el tiempo total de duración del programa desde la instrucción goto MAIN hasta goto FIN, asumiendo un cristal externo de 4 MHz conectado al PIC18F?

9. Analice el siguiente programa:

```
positivo EQU 0x20
negativo EQU 0x21

        org 0x00
        goto MAIN
        org 0x20
MAIN:
        movlw 0xFD
        movwf negativo
        movlw .10
        addwf negativo,W
        movwf positivo
FIN:
        goto FIN
        END
```

¿Qué valor toma la dirección 0x20 o positivo al finalizar el programa?

Respuestas

Respuestas del capítulo 4:

http://bit.ly/1OvaOxE

Capítulo 5. Rutinas típicas en lenguaje ensamblador

En las aplicaciones típicas haciendo uso de los microcontroladores es común el requerimiento de una serie de programas en lenguaje ensamblador que permitan realizar operaciones clásicas, como las rutinas de retardo de tiempo, conversión de bases binaria BCD, la división de dos números, manejo de un teclado matricial, control de LED, Displays 7 segmentos y pulsadores.

Toda aplicación, por más simple o compleja, requiere de un planteamiento o diagrama general que posibilite, en primer lugar, ordenar la idea básica del problema y identificar las variables y los procesos más importantes, y luego generar una secuencia lógica de la solución. Esta metodología, conocida como «diagrama de flujo», será abordada en este capítulo.

El presente capítulo busca orientar la forma de planteamiento de soluciones básicas con microcontroladores y generar un código en ensamblador para el PIC18F4550 que puede ser reutilizado por el lector para producir programas más complejos, al mismo tiempo que servirá para los ejemplos usados en el libro más adelante.

5.1 Diseño de un diagrama de flujo

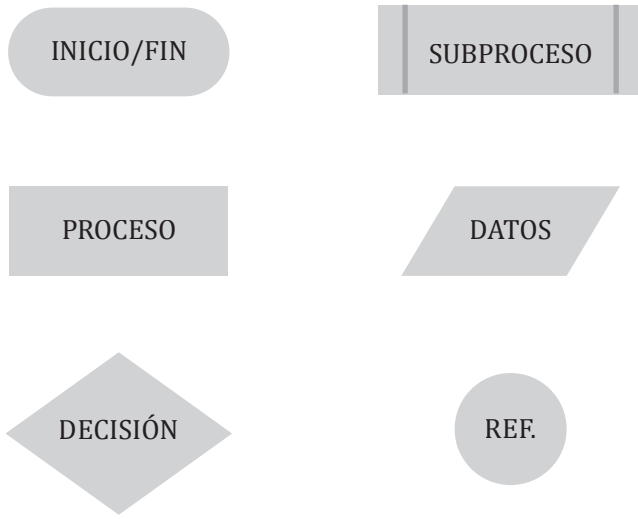
Un programador de sistemas embebidos o, más específicamente, de *firmware*, por lo general diseña soluciones a un determinado problema de ingeniería. Para poder comenzar creando la solución, es fundamental comprender la problemática íntegramente antes de comenzar a escribir la primera línea de instrucción. En esta etapa, lo común es definir un diagrama de bloques del sistema, indicando las entradas y salidas de la propuesta de solución y las interacciones entre estas.

Una vez que el problema es un tema conocido por el programador, es posible comenzar a definir un esquema para solucionar el problema a nivel de código. Esta solución se conoce como «algoritmo», que básicamente es una secuencia de operaciones que define la forma como se van a tratar las entradas del sistema para producir la o las salidas desde un punto de vista computacional. El algoritmo se compone de una secuencia de pasos computacionales que transforman la entrada del sistema en la salida. Por lo general, el algoritmo se representa en un pseudocódigo que se parece bastante al lenguaje ANSI C. Por otro lado, otro método bastante claro y que se utiliza en cualquier lenguaje de programación es el uso de diagramas de flujo. Esta metodología se basa en gráficos que especifican de manera secuencial y coherente los pasos que el algoritmo debe seguir.

Un diagrama de flujo se compone de símbolos clásicos que determinan una función típica. Por ejemplo, en el gráfico 5.1 se muestran los símbolos típicos de un diagrama de flujo básico. El primer elemento es el Terminal, que puede ser de dos tipos: Inicio o Fin. El primero delimita el inicio del programa, mientras que el segundo indica dónde este se acaba. El proceso sirve para indicar los nuevos valores que adquieren las variables o las ecuaciones que serán ejecutadas. Un bloque de decisión posee una entrada y dos salidas que dependen de una pregunta. De acuerdo con la respuesta (verdadera o

falsa), la ejecución de otros procesos puede cambiar de ruta en el diagrama. El subproceso involucra un conjunto de tareas que pueden estar asociadas a una función específica. Para efectos de las aplicaciones vistas, un subproceso puede ser una macro, o una rutina específica. El bloque de datos delimita las entradas al sistema y las salidas de este para su visualización. Finalmente, la referencia sirve para conectar algunos puntos del diagrama hacia la parte superior u otras posiciones que no siguen una secuencia descendente, o cuando el diagrama es muy grande y debe continuar en otra hoja.

Gráfico 5.1. Elementos de un diagrama de flujo



Elaboración propia.

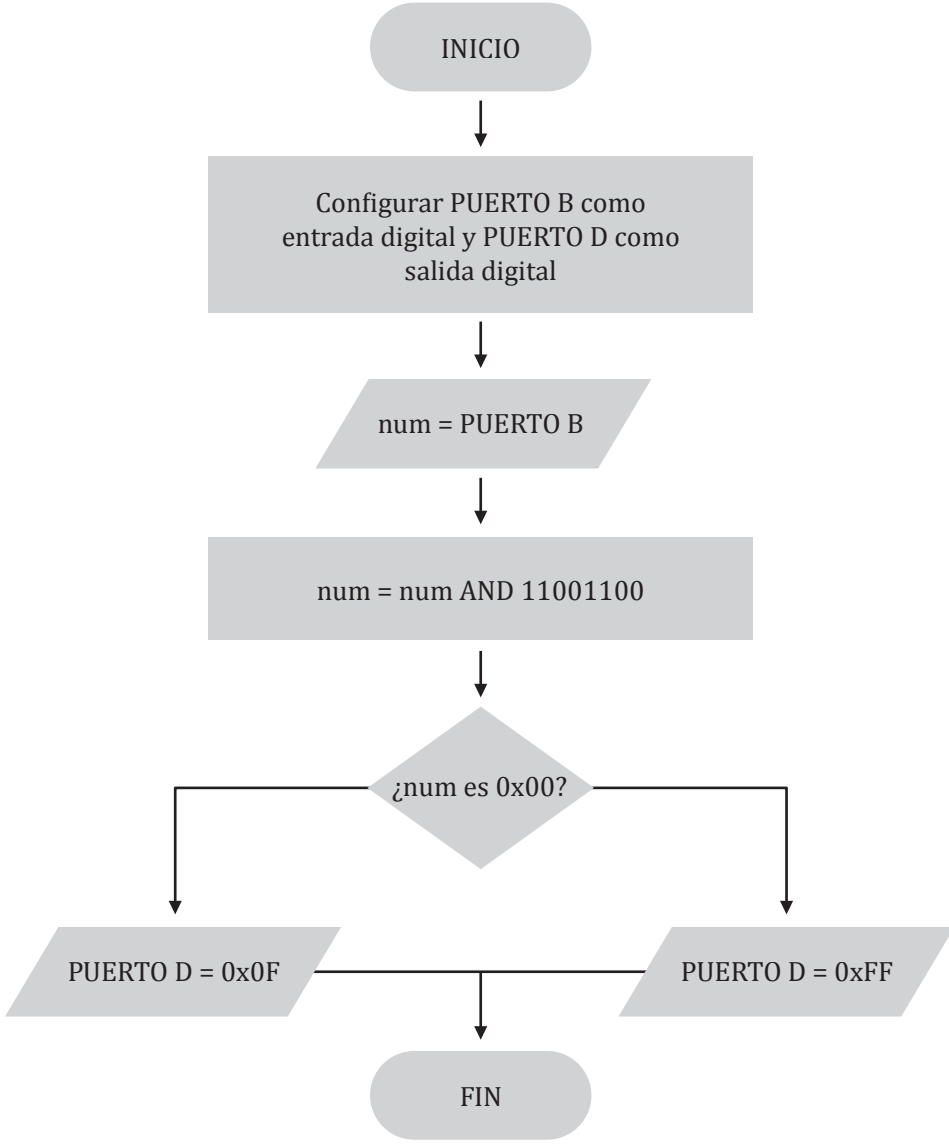
En el caso de que se desee desarrollar una aplicación en la que se lea un número binario del PUERTO B configurado como entrada digital y el valor leído realice la operación AND con la cifra 11001100_2, si el resultado es 0x00, el PUERTO D tomará el valor 0x0F, y si el resultado es diferente de 0x00, el PUERTO D tomará el valor 0xFF. Entonces, el diagrama de flujo sería como se muestra en el gráfico 5.2.

Como se puede observar, los bloques del diagrama de flujo se conectan por flechas, cada flecha indica el orden de ejecución de los procesos. El diagrama de flujo es bastante simple y permite entender los pasos que se deben realizar en el programa. Cada paso corresponde a un bloque del diagrama. Las etapas que se pueden determinar a partir del diagrama de flujo son los siguientes:

Paso 1: configurar todos los pines del PUERTO B como entrada digital y todos los pines del PUERTO D como salida digital.
Paso 2: leer el valor del PUERTO B en el registro num.
Paso 3: realizar la operación AND entre num y el valor binario 11001100_2. El resultado se almacenará en el registro num.
Paso 4: evaluar si el registro num es 0x00.
Paso 5: de ser verdadera la respuesta, el PUERTO D se cargará con el valor de salida 0x0F.
Paso 6: de ser falsa la respuesta, el PUERTO D se cargará con el valor de salida 0xFF.
Paso 7: fin del programa.

La estructura de los programas que se tomará para los ejemplos de este capítulo se muestra en el programa 5.1. Aquí únicamente se indicarán los archivos de cabecera a incluir en el programa, las directivas que declaran los alias, arreglos de datos, el vector de Reset, los vectores de interrupciones y el inicio del programa.

Gráfico 5.2. Diagrama de flujo de un problema simple de manipulación de puertos



Elaboración propia.

Programa 5.1. Modelo de organización del programa en lenguaje ensamblador a seguir en los ejemplos del libro

```
#include <P18F4550.INC>
;Directivas de Declaración de alias
        org 0x00
        goto MAIN
        org  0x08
        retfie          ;Vector de interrupción de alta prioridad
        org  0x18
        retfie          ;Vector de interrupción de baja prioridad
        org 0x20
MAIN:
        ;Instrucciones del programa
        .
        .
        .
        END
```

El programa 5.2 muestra la implementación del diagrama de flujo presentado en el gráfico 5.2. En él se puede observar cómo el orden del programa se encuentra directamente relacionado con los pasos establecidos.

Programa 5.2. Código solución del problema planteado en el diagrama de flujo del gráfico 5.2

```
#include<P18F4550.INC>
num EQU 0x20  ;Declaración del alias num en la dirección 0x20 de la memoria SRAM
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x20
MAIN:                       ;PASO 1
        clrf TRISD          ;Todos los pines del Puerto D como salida.
        setf TRISB          ;Todos los pines del Puerto B como entrada.
        movf  PORTB,W       ;PASO 2
        movwf num,0         ;Lectura en num del Puerto B.
        movlw b'11001100'   ;PASO 3
```

```
        andwf num,f,0          ;Operación AND entre num y el valor binario
        movlw 0x00             ;11001100_2. El resultado se guarda en num.
        cpfseq num,0           ;PASO 4
        goto  FALSO            ;Se compara num con 0 y se evalúa si es verdadero
                               ;o falso.
        goto  VERDADERO
VERDADERO:
        movlw 0x0F             ;PASO 5
        movwf LATD             ;Condición verdadera, se carga el Puerto D a 0x0F.
        goto  FIN
FALSO:
        setf  LATD             ;PASO 6
        goto  FIN              ;Condición falsa, se carga el Puerto D con 0xFF
FIN
        goto  FIN              ;PASO 7
        END                    ;Fin del programa, bucle indefinido
```

5.2 Rutinas básicas de retardo de tiempo

Las rutinas de retardo de tiempo son fundamentales para poder realizar aplicaciones con el microcontrolador en las cuales el usuario pueda visualizar la respuesta del programa. Por ejemplo, en un proceso tan simple como prender y apagar un LED conectado al PIN RB0 del PUERTO B, es necesario el uso de rutinas de retardo. Como ejemplo, se tiene un proceso que se realiza de la siguiente manera:

```
BUCLE:
        bcf LATB,0      ;Poner el pin RB0 a nivel '0' (LED apagado)
        bsf LATB,0      ;Poner el pin RB0 a nivel '1' (LED encendido)
        goto BUCLE      ;Se repite el proceso.
```

¿Cuál es el principal inconveniente de esta rutina? Principalmente que, si se desea que el usuario pueda visualizar el encendido y el apagado del LED, esto será imposible. La persona visualizará al LED continuamente encendido, ya que cada instrucción se ejecuta a una velocidad tan rápida que el ojo humano no podrá detectar los cambios.

Para efectos de este capítulo, se va a trabajar con una frecuencia de reloj externa de 4 MHz. Para esta velocidad, el periodo de reloj es de 0.25 µsegundos. Como se sabe, el tiempo mínimo de ejecución de una instrucción es de 4 periodos de reloj, es decir, 1 µsegundo. Esto implica que la instrucción bcf LATB,0 pondrá al LED apagado durante 1 µsegundo. Luego, la instrucción bsf LATB,0 encenderá al LED durante 1 µsegundo. Posteriormente, la rutina goto BUCLE tardará 2 µsegundos en ejecutarse (por ser instrucción de salto), con lo cual el LED permanecerá encendido por 3 µsegundos en total. Esto implica que el LED estará en prendido durante 3 µsegundos y apagado durante 1 µsegundo. El periodo de

encendido y apagado es de 4 µsegundos, con lo cual se tiene una frecuencia efectiva de 250 kHz. Esto es muy rápido para que el ojo humano pueda notar los cambios. Como se sabe, la máxima frecuencia de oscilación que puede notar la vista de una persona es de 50 Hz.

¿Cómo se puede solucionar, entonces, el problema de visualización de encendido y apagado del LED? Pues simplemente se debe agregar una rutina de retardo de tiempo que mantenga al LED apagado y encendido durante un lapso lo suficientemente grande como para que el ojo humano pueda notar los cambios. La nueva rutina de encendido y apagado lucirá de la siguiente manera:

```
BUCLE:
        bcf LATB,0              ;Poner el pin RB0 a nivel '0' (LED apagado).
        call RETARDO_500MS      ;Se invoca a una rutina que tarda 500 milisegundos
                                ;en retornar.
        bsf LATB,0              ;Poner el pin RB0 a nivel '1' (LED encendido)
        call RETARDO_500MS      ;Se invoca a una rutina que tarda 500 milisegundos
                                ;en retornar.
        goto BUCLE              ;Se repite el proceso.
```

Entonces, el usuario podrá visualizar el LED prendido por 0.5 segundos y apagado por el mismo tiempo, lo cual sí es detectable por el ojo humano. A continuación, se crearán rutinas para generar retardos de tiempo más pequeños y luego lograr obtener retardos en el orden de los segundos. Ahora se analizará el programa 5.3.

Programa 5.3. Rutina de retardo de 1 milisegundo aproximado

```
temp EQU 0x20
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x20
MAIN:
        clrf TRISB
BUCLE:
        bcf LATB,0              ;Poner el pin RB0 a nivel '0' (LED apagado)
        call RETARDO_1MSEG
        bsf LATB,0              ;Poner el pin RB0 a nivel '1' (LED encendido)
        call RETARDO_1MSEG
        goto BUCLE              ;Se repite el proceso
```

```
RETARDO_1MSEG:
        movlw .100          ;1µseg
        movwf temp,0        ;1µseg
DELAY:
        nop                 ;1µseg
        nop                 ;2µseg
        nop                 ;3µseg
        nop                 ;4µseg
        nop                 ;5µseg
        nop                 ;6µseg
        nop                 ;7µseg
        decfsz temp,f,0     ;8µseg
        goto  DELAY         ;10µseg
        return
        END
```

De la rutina RETARDO_1MSEG se puede notar lo siguiente:

1. Se carga el registro temp con el valor decimal .100 (este proceso demora 2 µsegundos).
2. Se ejecutan 7 instrucciones nop (que genera un retardo constante de 7 µsegundos).
3. La instrucción decfsz temp,f,0 demorará 1 µsegundo las primeras 99 veces que se ejecute.
4. la instrucción goto DELAY tarda 2 µsegundos en ejecutarse.

En total, entre el primer nop y la instrucción goto DELAY existen 10 µsegundos de retardo. Si este proceso se repite 100 veces, se tendrá un retardo equivalente de $100 \times 10\mu seg = 1000\mu seg = 1mseg$.

Sin embargo, hay un pequeño detalle. Al ingresar a la rutina RETARDO_1MSEG se ejecutan dos instrucciones adicionales (movlw .100, movwf temp,0), las cuales generan un retardo de 2 µsegundos. A esto hay que sumar el hecho de que la instrucción return toma 2 µsegundos más. Además, la instrucción call RETARDO_1MSEG (2 µsegundos), generando un retardo total de 6 µsegundos sobre el milisegundo generado por la rutina DELAY.

Como se puede observar, este tipo de retardo no es preciso. Las inexactitudes se generan porque siempre se van a requerir algunas instrucciones adicionales, tales como return, goto o las instrucciones que inicializan el valor de registro. A esto se debe sumar el hecho de que este tipo de rutina de retardo mantiene ocupado al procesador, el cual podría realizar otros procesos durante cada milisegundo de espera. Es por esta razón que este tipo de retardos por *firmware* se utilizan para aplicaciones en las cuales no se requiere exactitud, como es el caso de prender y apagar un LED, o la eliminación de rebotes en teclas pulsadas. Para efectos de aplicaciones que necesitan retardos de tiempo muy precisos se utilizan los módulos de temporización del microcontrolador combinado con interrupciones, tema que será abordado más adelante en el libro.

En el programa 5.4 se muestra una lógica para generar un retardo de tiempo en el orden de los milisegundos usando como base un cristal externo de 4 MHz. Se va a analizar el código para calcular el retardo total.

Programa 5.4. Rutina de retardo de tiempo para valores en el orden de los milisegundos y segundos

```
RETARDO:
        movlw .2            ;1us  TOTAL:2us
        movwf aux1,0        ;1us
RET1:
        movlw .45           ;1us  TOTAL:(aux1)x2us = 4us
        movwf aux2,0        ;1us
RET2:
        movlw .200          ;1us  TOTAL:(aux1)(aux2)x2us = 180us
        movwf aux3,0        ;1us
RET3:
        decfsz aux3,f,0     ;1us  TOTAL:aux3 x aux2 x aux1 x 1us = 18ms
        goto  RET3          ;2us  TOTAL:aux3 x aux2 x aux1 x 2us = 36ms
        decfsz aux2,f,0     ;1us  TOTAL:aux2 x aux1 x 1us = 90us
        goto  RET2          ;2us  TOTAL:aux2 x aux1 x 2us = 180us
        decfsz aux1,f,0     ;1us  TOTAL:aux1 x 1us  = 2us
        goto  RET1          ;2us  TOTAL:aux1 x 2us  = 4us
        return              ;2us  TOTAL:2us  = 2us
```

Como se puede observar, esta rutina se compone de 13 instrucciones e interactúa con 3 registros: aux1, aux2 y aux3. Los valores iniciales de estas tres variables van a definir el tiempo de retardo total. En el ejemplo aux1 = .2, aux2 = .45 y aux3 = .200. En el gráfico 5.3 se puede observar el conjunto total de retardos que genera cada instrucción.

Gráfico 5.3. Retardos de tiempo generados por cada instrucción de la rutina

N.° Instrucción	Instrucción	Tiempo de retardo	Tiempo total
1	movlw . 2	1µseg	1 µseg
2	movwf aux1,0	1µseg	1 µseg
3	movlw .45	(aux1)x1 µseg = 2 µseg	2 µseg
4	movwf aux2,0	(aux1)x1 µseg = 2 µseg	2 µseg
5	movlw .200	(aux1)x(aux2)x1 µseg	90 µseg
6	movwf aux3,0	(aux2)x(aux2)x1 µseg	90 µseg
7	decfsz aux3,f,0	(aux3)x(aux2)x(aux1)x1 µseg	18 mseg
8	goto RET3	(aux3)x(aux2)x(aux1)x2 µseg	36 mseg
9	decfsz aux2,f,0	(aux2)x(aux1)x1 µseg	90 µseg
10	goto RET2	(aux2)x(aux1)x2 µseg	180 µseg
11	decfsz aux1,f,0	(aux1)x 1µseg	2 µseg
12	goto RET1	(aux1)x 2 µseg	4 µseg
13	Return	2 µseg	2 µseg

Elaboración propia.

Si se observa detenidamente, son las instrucciones 7 y 8 las que más aportan al retardo, por ser las que más veces se repiten en el programa. Entre estas dos instrucciones se tienen 54 milisegundos de retardo. El resto de las instrucciones aportan un total de 464 µsegundos, lo que implica un retardo total de 54.464 milisegundos.

Si se quisiera llevar el tiempo total de retardo a una ecuación, se podría representar de la siguiente manera:

$$\text{Re}tardo = 4\mu seg + aux1(5\mu seg) + aux1 \times aux2(5\mu seg) + aux1 \times aux2 \times aux3(3\mu seg)$$

Para obtener un retardo aproximado de 500 milisegundos con un cristal de 4 MHz, se recomienda introducir esta ecuación en un cuadro de Microsoft Excel, donde se podrán calcular los valores de los registros aux1, aux2 y aux3 de manera rápida para el retardo requerido. Por ejemplo, con aux1 = .18, aux2 = .44 y aux3 = .209, se logra un retardo total de 500.638 milisegundos, que es un valor aproximado.

5.3 Conversión de bases

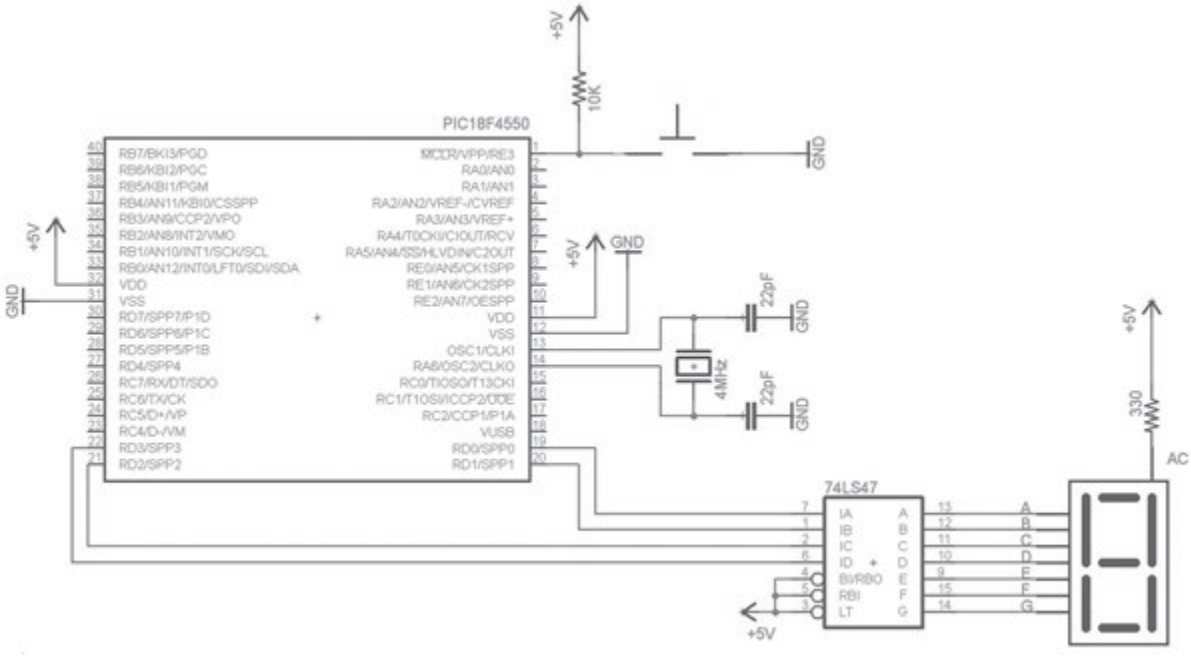
Las rutinas de conversión de bases son fundamentales para el tratamiento con cifras en formato BCD o ASCII. Esto es muy importante cuando se trabaja con módulos LCD alfanuméricos, puertos RS232 o múltiples dispositivos que manejan estos sistemas de numeración.

A continuación se presenta una serie de ejemplos aplicados para demostrar la funcionalidad de las rutinas de conversión de bases.

Ejemplo 5.1

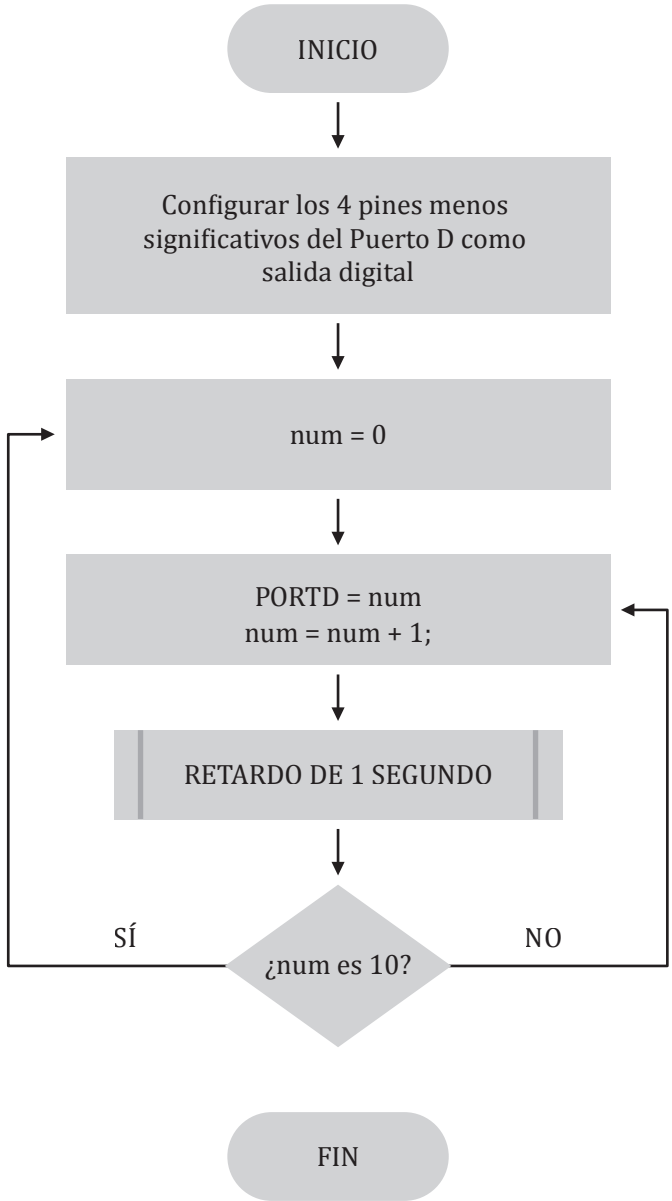
En este primer ejemplo se muestra un diagrama esquemático del PIC18F4550 conectado a un decodificador BCD-7 segmentos 74LS47 a través de los cuatro pines menos significativos del Puerto D, tal como se muestra en el gráfico 5.4. El circuito integrado 74LS47 se encarga de transformar un código BCD de cuatro cifras en el formato que permite mostrar el dígito decimal en un Display 7 segmentos de tipo ánodo común. El microcontrolador, entonces, se encargará de generar la secuencia hexadecimal 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x00, 0x01….0x09,… cada segundo. En el gráfico 5.5 se muestra el diagrama de flujo del programa propuesto, y en el programa 5.5 se muestra el código con la solución.

Gráfico 5.4. Circuito esquemático del programa de conteo BCD



Elaboración propia.

Gráfico 5.5. Diagrama de flujo del programa de conteo binario



Elaboración propia.

Programa 5.5. Algoritmo para la generación de una secuencia de conteo binaria de 0 a 9

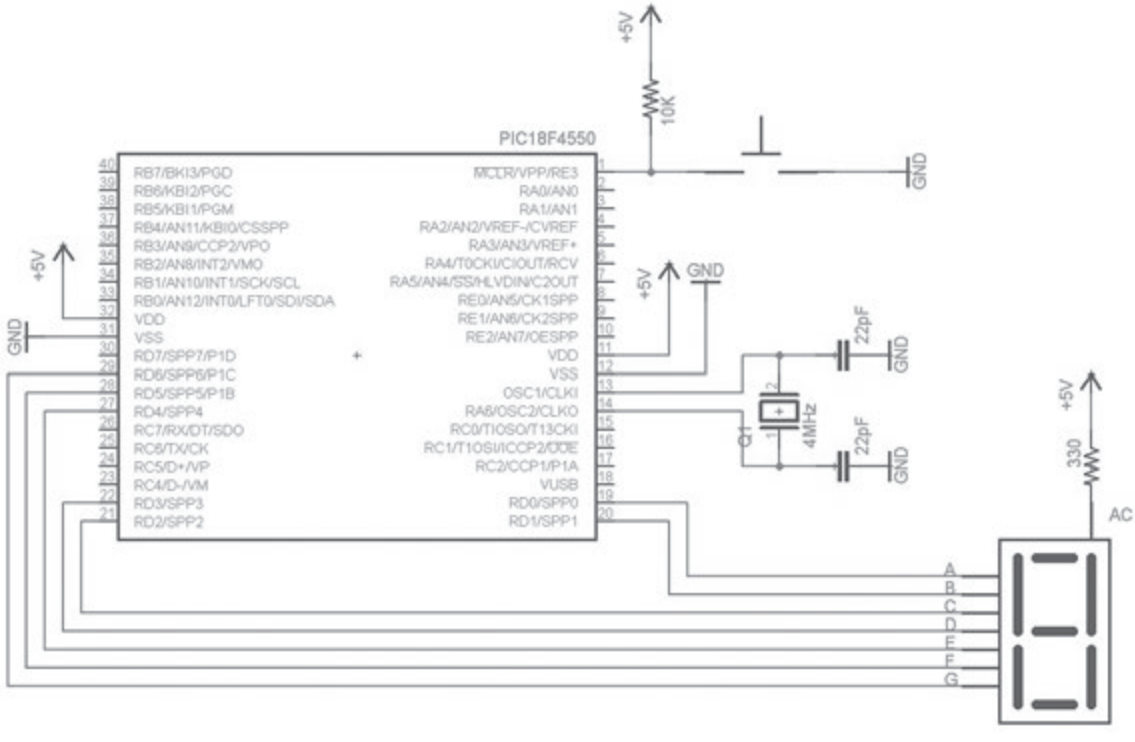
```
#include<P18F4550.INC>
cblock 0x20
        aux1
        aux2
        aux3
        cont
endc

        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x200
MAIN:
        movlw 0xF0              ;Se configuran los cuatro pines menos significativos del
                                 ;puerto D como salida digital.
        movwf TRISD
        clrf cont               ;el registro con dirección SRAM 0x23 se carga con 0x00.
INICIO:
        movf cont,W             ;Se carga el contenido de la dirección absoluta 0x23 a cont.
        movwf LATD              ;Se carga la cifra BCD en el Puerto D.
        call  RETARDO500MS      ;Se genera un retardo de 0.5 segundos
                                ;aproximadamente.
        call  RETARDO500MS      ;Se genera un Segundo retardo de 0.5 segundos
                                ;aproximadamente
        incf cont,f             ;Se incrementa el valor de la dirección absolouta 0x23 en 1.
        movlw .10               ;Se realiza la comparación del valor de la dirección 0x23
                                ;con .10
        cpfseq cont             ;Si son iguales hay que empezar el conteo desde 0.
        goto  INICIO            ;Si son diferentes se debe seguir incrementando la dirección
                                ;0x23 en 1, mostrando su valor en el Puerto D.
        clrf cont
        goto  INICIO
RETARDO500MS:
        movlw .18
        movwf aux1,0
RET1:
        movlw .44
        movwf aux2,0
```

```
RET2:
        movlw .209
        movwf aux3,0
RET3:
        decfsz aux3,f,0
        goto  RET3
        decfsz aux2,f,0
        goto  RET2
        decfsz aux1,f,0
        goto  RET1
        return
        END
```

Ejemplo 5.2

Posteriormente, se realizará un ejemplo parecido al mostrado en el ejemplo 5.1, con la diferencia de que la conversión BCD – 7 segmentos se realiza en el código del programa mediante el uso del puntero de tabla. Si se analiza el circuito esquemático propuesto en el gráfico 5.6, se observará que la línea RD0 controla el segmento A del display, la línea RD1, el segmento B, y así hasta el pin RD6, que controla el segmento G. El display 7 segmentos es de tipo ánodo común, lo cual implica que cada segmento se encenderá con un nivel lógico '0', mientras que permanecerá apagado con el nivel lógico '1'. En el gráfico 5.7 se muestran los valores que deben asignarse a cada segmento para producir una secuencia numérica del 0 al 9.

Gráfico 5.6. Diagrama esquemático del programa de conversión hexadecimal – 7 segmentos



Elaboración propia.

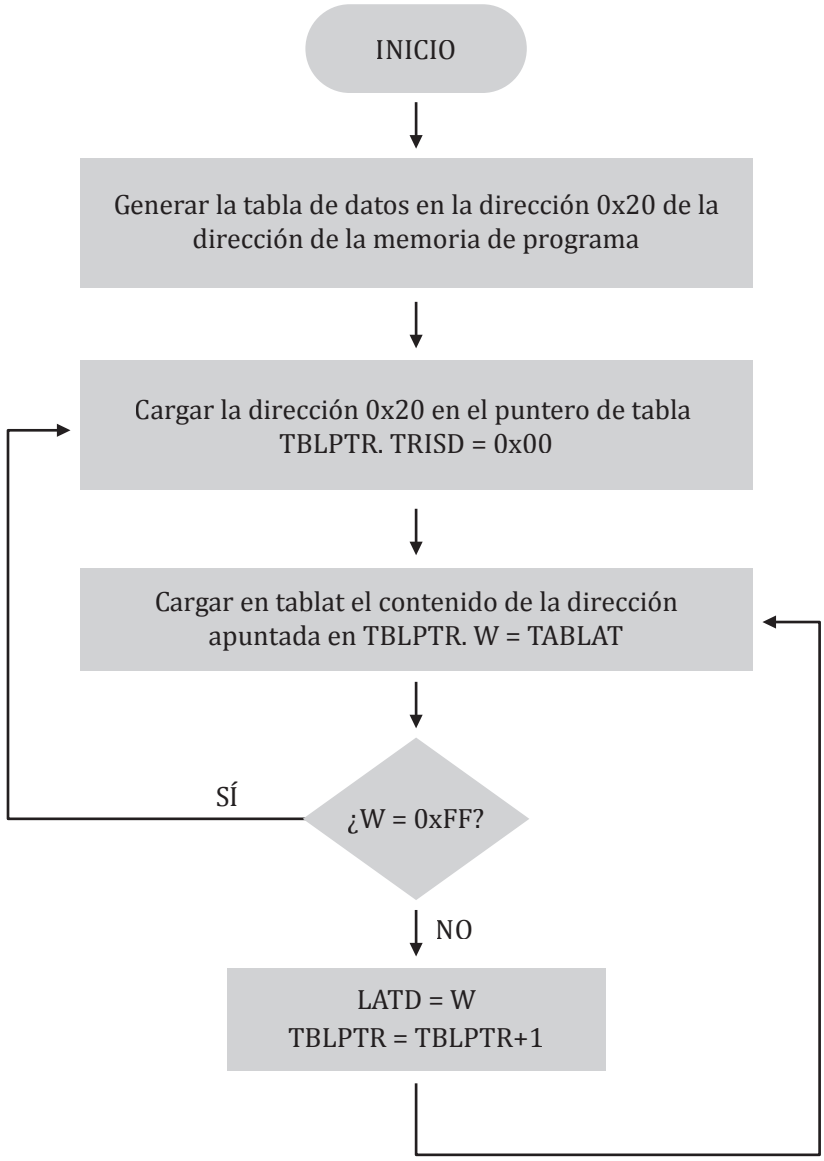
Gráfico 5.7. Combinaciones binarias para producir cifras del 0 al 9 en el Display 7 segmentos tipo ánodo común

Segmento G (RD1)	Segmento F (RD1)	Segmento E (RD1)	Segmento D (RD1)	Segmento C (RD1)	Segmento B (RD1)	Segmento A (RD0)	Cifra 7 seg
1	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1
0	1	0	0	1	0	0	2
0	1	1	0	0	0	0	3
0	0	1	1	0	0	1	4
0	0	1	0	0	1	0	5
0	0	0	0	0	1	1	6
1	1	1	1	0	0	0	7
0	0	0	0	0	0	0	8
0	0	1	1	0	0	0	9

Elaboración propia.

El procedimiento es bastante simple. Se crea un arreglo de bytes en la memoria de programa con los diez valores presentados en el gráfico 5.7. Este arreglo parte en la dirección 0x20, la cual es indicada a través de una etiqueta de nombre SIETESEG. Posteriormente, se carga el puntero de tabla TBLPTR con la dirección inicial de la tabla y se realiza la lectura de los elementos de la tabla uno por uno. Antes de mostrar un elemento en el Puerto D, se analiza si este contiene el valor 0xFF. De ser afirmativa esta condición, entonces se habrá llegado a leer el último byte de la tabla, por lo cual se debe inicializar la lectura desde el primer elemento. En el gráfico 5.8 se muestra el diagrama de flujo de este programa.

Gráfico 5.8. Diagrama de flujo del programa 5.6



Elaboración propia.

Programa 5.6. Algoritmo conversor de código hexadecimal a siete segmentos

```
#include<P18F4550.INC>
cblock 0x20
        aux1
        aux2
        aux3
        cont
endc
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x20                    ;Dirección base 0x20.
                                    ;Creación de una tabla de 10 elementos.

SIETESEG: db 0x40,0x79,0x24,0x30,0x19,0x12,0x03,0x78,0x00,0x18
        org 0x200                   ;Dirección de programa 0x200.
MAIN:
        clrf  TRISD                 ;Todos los pines del Puerto D son salida.
        movlw UPPER SIETESEG        ;Se cargan los 5 bits más significativos
        movwf TBLPTRU               ;en el registro TBLPTRU.
        movlw HIGH  SIETESEG        ;Se cargan los bits del 15 al 8 de la
        movwf TBLPTRH               ;dirección 0x200 en el registro TBLPTRH.
        movlw LOW   SIETESEG        ;Se cargan los bits 7 al 0 de la dirección
        movwf TBLPTRL               ;0x200 en el registro TBLPTRL.
INICIO:
        TBLRD*+                     ;Se carga el valor de la dirección apuntada
                                    ;por TBLPTR en el registro TABLAT.
        movf  TABLAT,W              ;Luego, se incrementa TBLPTR.
        movwf cont                  ;Se guarda en cont el valor de TABLAT.
        movlw 0xFF                  ;Se carga el registro W con 0xFF y se
        xorwf cont,W                ;compara con cont, realizando la operación
                                    ;XOR.
        btfsc STATUS,Z              ;Se consulta si el bit Z = 0. Esto implica
                                    ;que no son iguales.
        goto  MAIN                  ;Si Z=1 son iguales por tanto se inicia
                                    ;nuevamente el programa.
        movf cont,W                 ;Se carga el valor de cont en W.
        movwf LATD                  ;Se envía al Puerto D.
        call  RETARDO500MS          ;Retardo de 0.5 segundos
        call  RETARDO500MS          ;Retardo de 0.5 segundos
        goto  INICIO                ;Se repite la lectura de la tabla con el
                                    ;nuevo valor del puntero.
```

```
RETARDO500MS:
        movlw .18                ;1us  TOTAL:2us
        movwf aux1,0             ;1us
RET1:
        movlw .44                ;1us  TOTAL:(aux1)x2us = 4us
        movwf aux2,0             ;1us
RET2:
        movlw .209               ;1us  TOTAL:(aux1)(aux2)x2us = 180us
        movwf aux3,0             ;1us
RET3:
        decfsz aux3,f,0          ;1us  TOTAL:aux3 x aux2 x aux1 x 1us = 18ms
        goto  RET3               ;2us  TOTAL:aux3 x aux2 x aux1 x 2us = 36ms
        decfsz aux2,f,0          ;1us  TOTAL:aux2 x aux1 x 1us  = 90us
        goto  RET2               ;2us  TOTAL:aux2 x aux1 x 2us  = 180us
        decfsz aux1,f,0          ;1us  TOTAL:aux1 x 1us  = 2us
        goto  RET1               ;2us  TOTAL:aux1 x 2us  = 4us
        return                   ;2us  TOTAL = 2us
        END
```

Ejemplo 5.3

La conversión binario-BCD es una aplicación bastante necesaria en la mayoría de programas que requieren realizar cálculos matemáticos y mostrar los resultados en Displays 7 segmentos, módulos LCD alfanuméricos o gráficos, inclusive en dispositivos que manejan comunicación serial asíncrona, como módems.

De esta manera, la intención principal es desarrollar una rutina que permita convertir cualquier cifra de 8 bits en tres cifras BCD. Por ejemplo, la cifra binaria 10000001_2 (129 en decimal) en BCD sería $0001\ 0010\ 1001_2$. Una vez convertida en formato BCD es mucho más sencillo transformar cada dígito en formato ASCII u otro estándar necesario para configurar algún dispositivo.

El proceso para convertir una muestra de 8 bits en un formato BCD consiste en tomar la cifra binaria y someter a todos sus bits a un procedimiento de transformación, según los siguientes pasos:

1. Desplazar un bit a la izquierda la cifra binaria desde la posición LSB a la MSB.
2. Si alguna de las cifras BCD generadas (4 bits) es mayor a cuatro, se le deberá sumar el valor de tres.
3. Repetir los pasos del 1 al 2 hasta que todos los bits de la cifra original se hayan desplazado.

Por ejemplo, en el gráfico 5.9 se puede observar el proceso de conversión de la cifra binaria 1011110_2 (0xBE) en el valor BCD $0001\ 1001\ 0000_2$.

Gráfico 5.9. Ejemplo de aplicación de los pasos para la conversión binaria BCD

	BCD 2	BCD 1	BCD 0	Entrada binaria
INICIO				1 0 1 1 1 1 1 0
Bit 7			1	0 1 1 1 1 1 0
Bit 6			1 0	1 1 1 1 1 0
Bit 5			1 0 1 + 1 1 1 0 0 0	1 1 1 1 0
Bit 4		1	0 0 0 1	1 1 1 0
Bit 3		1 0	0 0 1 1	1 1 0
Bit 2		1 0 0	0 1 1 1 + 1 1 1 0 1 0	1 0
Bit 1		1 0 0 1 + 1 1 1 1 0 0	0 1 0 1 + 1 1 1 0 0 0	0
Bit 0	1	1 0 0 1	0 0 0 0	

Elaboración propia.

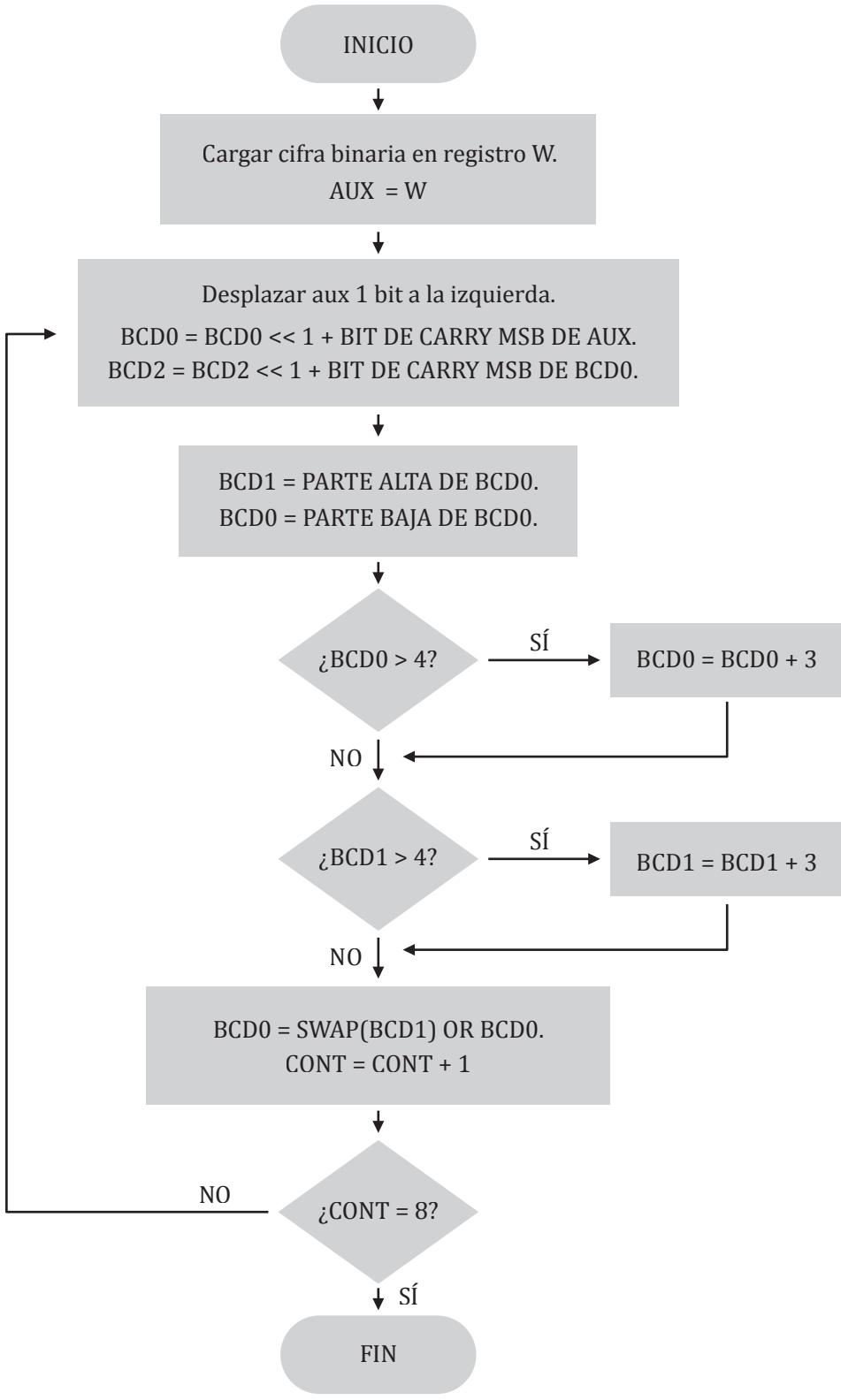
El procedimiento para realizar la conversión binaria hexadecimal en lenguaje ensamblador se muestra en el programa 5.7. El diagrama de flujo se muestra en el gráfico 5.10. El proceso comienza colocando el número binario (entrada binaria) en el registro aux. Este registro cumple con la función de representar la entrada binaria en el esquema del gráfico 5.9. De esta manera, aux se va desplazando un bit a la izquierda en cada iteración, por lo cual su bit MSB perdido, luego de ejecutar la instrucción rlcf, se convierte en el bit *Carry* del registro STATUS. Luego, cuando se corre a la izquierda el registro BCD0, esta toma como su nuevo bit LSB el valor del bit *Carry*, que corresponde al bit MSB perdido del registro aux[38].

Conforme los bits de BCD0 se van desplazando a la izquierda, el *nibble* más significativo de BCD0 se carga en el registro BCD1 para comparar cada valor equivalente con 5. Si uno de los registros es mayor que 4, entonces se le sumará el valor de 3. Luego, se carga el contenido del registro BCD1 con el *nibble* más significativo de BCD0 y se continúa con la secuencia de desplazamiento a la izquierda. Lo

38 Cfr. Pong Chu 2008.

mismo se realiza con el registro BCD2, solo que aquí el valor nunca será mayor que 4, ya que esta cifra solo puede ser 0, 1 o 2 (dado que la cifra binaria de entrada es de 8 bits).

Gráfico 5.10. Diagrama de flujo del convertidor binario BCD



Elaboración propia.

Programa 5.7. Conversión binaria a BCD

```
#include <P18F4550.INC>
cblock 0x20
        cont
        aux
        BCD2
        BCD1
        BCD0
endc
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x200
MAIN:
        movlw .139              ;Se carga el valor decimal .139.
        call  BIN_BCD           ;Se invoca a la rutina de conversión BCD.
        movff BCD0,0x30         ;Se guarda el dígito BCD menos significativo a la dirección
                                ;0x30.
        movff BCD1,0x31         ;Se guarda el dígito BCD alto a la dirección 0x31.
        movff BCD2,0x32         ;Se guarda el dígito BCD más significativo a la dirección
                                ;0x32
FIN:
        goto FIN

BIN_BCD:
        movwf aux,0             ;Se guarda el valor a convertir en aux.
        clrf  cont,0            ;cont=0x00.
        clrf  BCD0,0            ;BCD0=0x00
        clrf  BCD1,0            ;BCD1=0x00
        clrf  BCD2,0            ;BCD2=0x00
CONV:
        rlcf aux,f              ;Rotar a la izquierda cont (cifra original).
        rlcf  BCD0,f            ;Rotar a la izquierda BCD0 (carga el carry de cont en el
                                ;bit LSB).
        rlcf  BCD2,f            ;Rotar a la izquierda BCD2 (carga el carry de BCD0 en
                                ;el bit LSB).
;Cargar el nibble alto de BCD0 a BCD2 y analizar
        movf  BCD0,W            ;W = BCD0.
```

```
        movwf BCD1,0        ;BCD1 = BCD0.
swapf BCD1,f                ;Inversión de nibbles en BCD1.
        movlw 0x0F          ;W = 0x0F.
        andwf BCD1,f        ;BCD1 = BCD1 AND 0x0F.
        andwf BCD0,f        ;BCD0 = BCD0 AND 0x0F.
;Se pregunta si ya se llegaron a rotar a la izquierda los 8 bits
        movlw .7            ;W = .7.
        cpfslt cont,0       ;Salta si cont es menor a .7.
        return              ;Si cont = .7 retorna (FIN de la rutina).
;Averiguar si los nibbles de BCD0 y BCD1 son mayores que .4
        movlw .5            ;W = .5
        subwf BCD0,W        ;W = BCD0 - .5
        btfsc STATUS,C      ;Si BCD0>4 => Carry = 1
        call  SUMA3_BCD0    ;BCD0>4, hay que sumar 3
        movlw .5            ;BCD0 <5, se comprueba BCD1
        subwf BCD1,W        ;W = BCD1 - .5
        btfsc STATUS,C      ;Si BCD1>4 => Carry = 1
        call  SUMA3_BCD1    ;BCD1>4, hay que sumar 3
        swapf BCD1,f        ;Inversión de nibbles en BCD1
        movf  BCD1,W        ;W = BCD1
        iorwf BCD0          ;BCD0 = BCD1 OR BCD0
        incf  cont,f,0      ;cont = cont + 1
        goto  CONV          ;Se repite el proceso
SUMA3_BCD0:
        movlw .3            ;W = .3
        addwf BCD0,f,0      ;BCD0 = BCD0 + .3
        return              ;retorno del call
SUMA3_BCD1:
        movlw .3            ;W = .3
        addwf BCD1,f,0      ;BCD1 = BCD1 + .3
        return              ;retorno del call
        END
```

5.4 Operaciones matemáticas

Las operaciones matemáticas, específicamente las aritméticas, son muy importantes en el desarrollo de aplicaciones con el microcontrolador. Afortunadamente, el conjunto de instrucciones permite realizar una serie de operaciones básicas, como suma, resta, comparación de cifras y multiplicación. Estas operaciones son muy necesarias y se pueden ejecutar en un solo ciclo de instrucción. Sin embargo, queda pendiente una operación muy importante, como la división. No es común encontrar un microcontrolador de 8 bits de CPU que contenga una instrucción de división, por lo cual esta operación debe ser implemen-

tada combinando diferentes instrucciones. En el presente subcapítulo se muestran algunos algoritmos matemáticos sencillos junto con dos rutinas eficientes para la división aritmética de cifras binarias.

Ejemplo 5.4

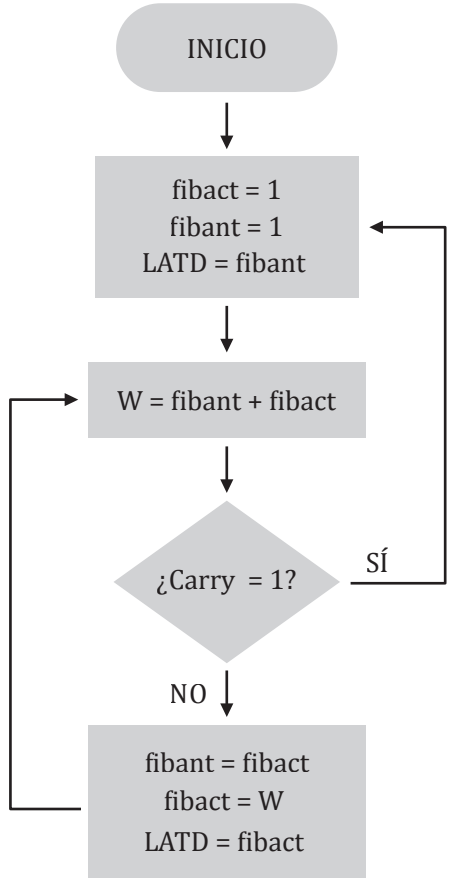
A continuación se ofrece un ejemplo en el cual se muestra la serie de Fibonacci con cifras de 8 bits presentadas en el puerto D del microcontrolador. La serie de Fibonacci es la siguiente: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377…… Como se puede ver, la secuencia empieza con 0,1, luego el siguiente elemento es la suma de los dos anteriores. Dado que en el microcontrolador la salida será visualizada en un puerto de 8 bits, se truncará la serie hasta el valor 233. En el gráfico 5.11 se muestra el diagrama de flujo de este programa. Se utilizan dos registros: fibact y fibant, el primero para almacenar el valor actual de la serie, y el segundo, el valor anterior. Ambas variables se inicializan con .1. Luego, fibact se incrementa con el valor de fibant, mientras que fibant toma el último valor de fibact. Este proceso se repite hasta que el valor de fibact sea mayor a 255, generando un desbordamiento en el bit *Carry*. Ante esta situación, la serie de Fibonacci se repite, cargando el valor de 1 en ambas variables, repitiéndose el proceso. En el programa 5.8 se muestra el algoritmo con la serie de Fibonacci implementada.

Programa 5.8. Algoritmo de la serie de Fibonacci

```
        cblock 0x20
                fibact
                fibant
        endc
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x20
MAIN:
        clrf  TRISD             ;Todos los pines del Puerto D como salida
        movlw .1                ;W = .1
        movwf fibact            ;fibact = .1
        movwf fibant            ;fibant = .1
        movwf LATD              ;LATD = .1
INICIO:
        movf  fibant,W          ;W = fibant
        addwf fibact,W          ;W = fibact + fibant
        btfsc STATUS,C          ;¿Hubo desbordamiento?
        goto  MAIN              ;Si hubo desbordamiento se salta a MAIN y se repite
                                ;la serie.
```

```
        movff fibact,fibant     ;fibant = fibact
        movwf fibact            ;fibact = W
        movwf LATD              ;LATD = fibact
        goto  INICIO            ;Se repite el proceso
END
```

Gráfico 5.11. Diagrama de flujo del programa que genera la serie de Fibonacci



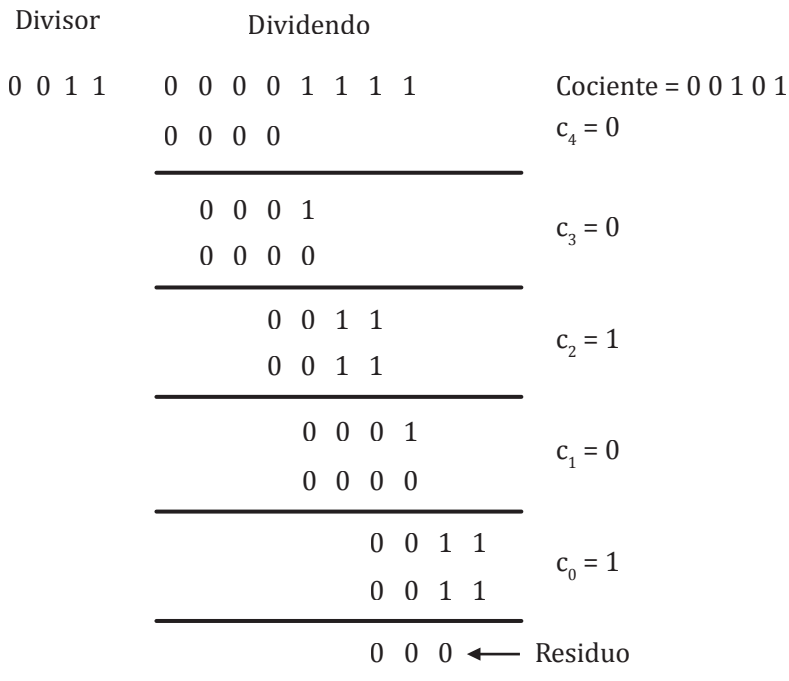
Elaboración propia.

Ejemplo 5.5

Un proceso aritmético sumamente importante es la división de dos números. El proceso más simple sería realizar restas sucesivas, pero esto implicaría que el tiempo de ejecución de la división dependería del valor numérico del divisor y del dividendo, lo cual haría ineficiente el proceso, ya que tendría una duración variable.

Un método más eficiente para realizar la división aritmética consiste en posicionar el divisor al lado del dividendo, completando este último con 4 ceros a la izquierda, tal como se muestra en el gráfico 5.12. Se toman los 4 bits más significativos (4 ceros) y se comparan con el divisor. Si el valor formado por estos 4 bits es menor al divisor, se resta 0000_2 y se deja caer el siguiente bit del dividendo, formando una nueva cifra de 4 bits. Ante esta situación, una variable cociente actualiza cada uno de sus bits (empezando de la posición MSB a la LSB) con '0'. Nuevamente se compara con el divisor. Si la cifra resulta mayor o igual al divisor, se resta el valor del divisor y nuevamente se deja caer el siguiente bit del dividendo (y de esta manera la variable cociente actualiza uno de sus bits con '1'). El proceso se repite hasta que el último bit en caer sea el LSB del dividendo. Cuando el proceso termina, la variable cociente tiene el resultado de la división y el valor residual de la última resta resulta ser el residuo. En el programa 5.9 se muestra el procedimiento en lenguaje ensamblador, además de un ejemplo de la división de .14 entre .5. El dividendo se debe colocar en el registro divdn y el divisor en el registro divsr antes de invocar a la rutina DIVISION_4BITS. La rutina DIVISION_4BITS retorna en el registro cocint el cociente de la división y en el registro resid el residuo del procedimiento aritmético[39].

Gráfico 5.12. Proceso de división de cifras de 4 bits



Elaboración propia.

39 Cfr. Pong Chu 2008, revisado y modificado por Salas 2014.

Programa 5.9. División binaria de números de cuatro bits

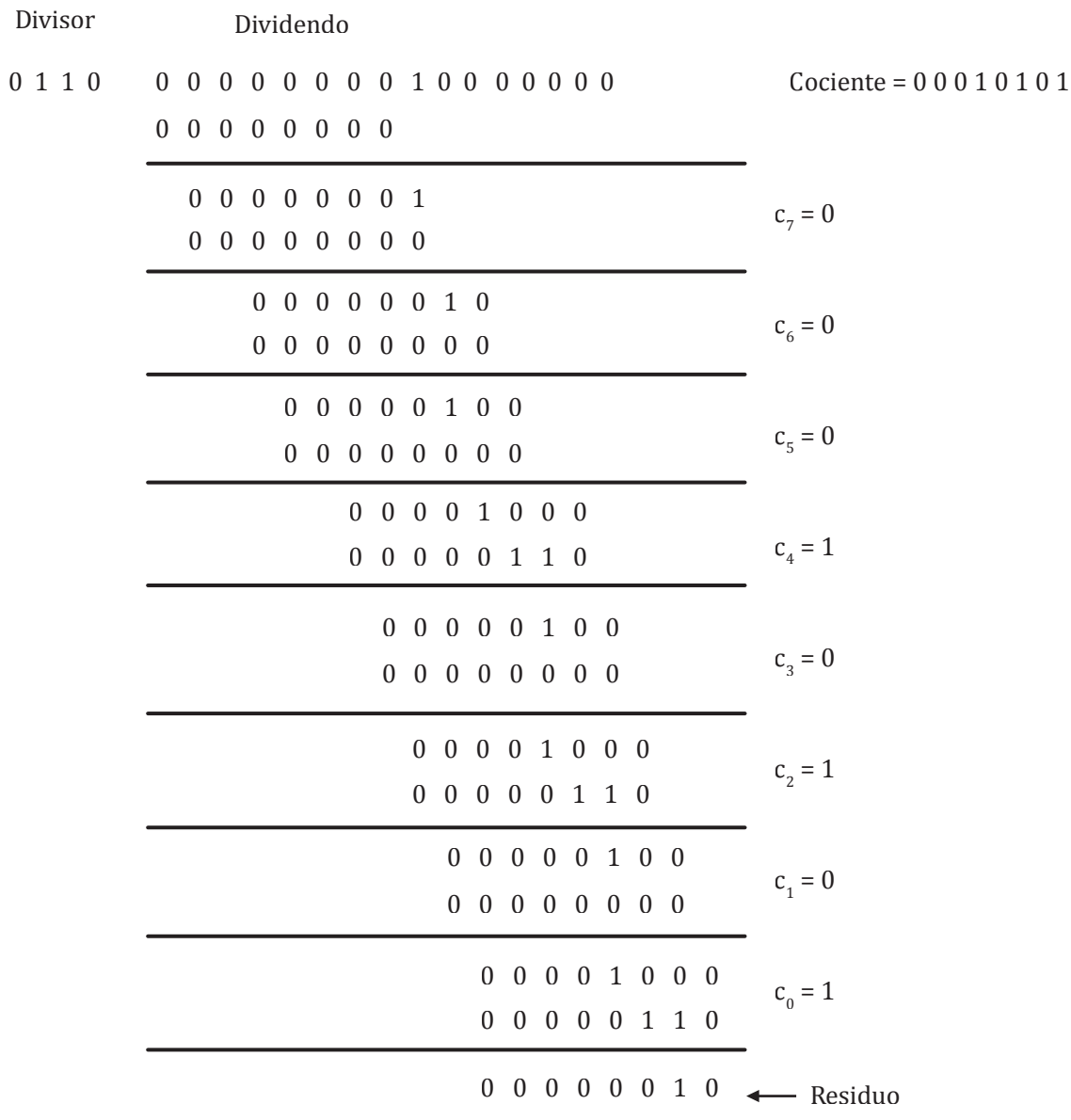
```
#include <P18F4550.INC>
cblock 0x20
        divsr
        divdnd
        cocint
        resid
        aux
        cont
endc
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x200
MAIN:
        movlw .14               ;Se carga en el dividendo.
        movwf divdnd            ;la cifra de prueba .14.
        movlw .5                ;Se carga en el divisor la
        movwf divsr             ;cifra de prueba .5.
        call  DIVISION_4BITS    ;Se invoca a la rutina de división
        movff cocint,0x30       ;Se guarda el cociente (.2) en la dirección 0x30 de
                                ;la SRAM.
        movff resid,0x31        ;Se guarda el residuo (.4) en la dirección 0x31 de
                                ;la SRAM.
FIN:
        goto FIN                ;Bucle indefinido;
DIVISION_4BITS:
        clrf   cont,0           ;Se inicializa el contador de bits en 0x00.
        clrf   cocint,0         ;Se inicializa el cociente en 0x00.
        movlw  0x00             ;Se compara el divisor con 0 para evitar.
        cpfseq divsr            ;realizar la división/0.
        goto   VERIFICA         ;Si no es 0 se salta a VERIFICA.
        clrf   cocint,0         ;Si el divisor es 0 se retorna cociente.
        clrf   resid,0          ;y residuo = .0.
        return
```

```
VERIFICA:
        movf divdnd,W           ;En VERIFICA se analiza si el dividendo.
        cpfsgt divsr            ;es mayor o igual al divisor.
        goto  DIVISION          ;Si es así, se salta a DIVISION.
        clrf  cocint,0          ;En caso contrario se retorna con el cociente
        clrf  resid,0           ;y el residuo = .0
        return
DIVISION:                       ;Proceso de DIVISION.
        bcf   STATUS,C          ;El Carry = 0 para no afectar el desplazamiento a
                                ;la izquierda.
        rlcf divdnd,f           ;Se rota un bit la izquierda el dividendo.
        movff divdnd,aux        ;aux = dividendo.
        movlw  0x0F             ;Se extrae pone a 0 el nibble más significativo.
        andwf divdnd,f          ;del dividendo.
        swapf aux,f             ;Se invierten los nibbles de aux
        movlw  0x0F             ;Se realiza la operación AND con 0x0F para
        andwf aux,f             ;rescatar el nibble menos significativo de aux.
        movf divsr,W            ;Se carga el divisor en W.
        cpfslt aux              ;Si aux<divisor salta.
        call   RESTA            ;Si aux>=divsr hay que restar aux = aux - divsr
        swapf aux,f             ;Se invierten los nibbles de aux.
        movlw  0xF0             ;Se rescata el nibble más significativo de
        andwf aux,W             ;aux. Luego, se realiza la operación OR con el
        iorwf divdnd,f          ;dividendo para recuperar el valor original
                                ;desplazado 1 bit.
        rlncf cocint,f          ;El registro cociente se un bit a la izquierda.
        incf cont,f             ;Se incrementa el contador de operaciones en 1.
        movlw  .4               ;Se compara el contador con .4. Si es .4 ya se
        cpfseq cont             ;terminó el proceso.
        goto   DIVISION         ;En caso que cont<4 se continua operando.
        swapf aux,W             ;aux contiene el residuo. Se invierten los nibbles.
        movwf resid             ;y se carga el valor en el registro resid.
        rrncf cocint,f          ;El cociente se ha desplazado a la izquierda 5
                                ;veces, solo debieron ser 4 veces.
        return                  ;Se desplaza una a la derecha y se retorna
RESTA:
        bsf  cocint,0           ;Si hay resta el bit q = 1.
        movf divsr,W            ;Se resta el divisor de aux.
        subwf aux,f             ;El resultado se guarda en aux y se retorna.
        return
        END
```

Ejemplo 5.6

El proceso de división binaria de 8 bits es muy similar al proceso de 4 bits, con la diferencia de que se tendrán nueve restas en total en vez de cinco. El divisor puede ser de cualquier valor de 8 bits. En el gráfico 5.13 se muestra el procedimiento de la división.

Gráfico 5.13. Proceso de división binaria para 8 bits de dividendo y divisor



Elaboración propia.

El procedimiento de división binaria de 8 bits es muy parecido al programa 5.11, con la diferencia de que se iteran nueve restas en vez de cinco. Nuevamente, los registros divsr y divdnd contienen al divisor y dividendo respectivamente, pero ahora estos pueden tomar cifras de hasta .255. Los registros cocint y resid contienen el cociente y divisor respectivamente, luego de ejecutarse la rutina DIVISION_8BITS.

Programa 5.10. División binaria de números de ocho bits

```
#include <P18F4550.INC>
cblock 0x20
        divsr
        divdnd
        cocint
        resid
        aux
        cont
endc
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x200
MAIN:
        movlw .248              ;Se carga en el dividendo
        movwf divdnd            ;la cifra de prueba .248.
        movlw .11               ;Se carga en el divisor la
        movwf divsr             ;cifra de prueba .11.
        call  DIVISION_8BITS    ;Se invoca a la rutina de división.
        movff cocint,0x30       ;Se guarda el cociente(.2) en la dirección 0x30 de
                                ;la SRAM.
        movff resid,0x31        ;Se guarda el residuo (.4) en la dirección 0x31 de
                                ;la SRAM.
FIN:
        goto FIN                ;Bucle indefinido

DIVISION_8BITS:
        clrf   aux,0            ;Se inicializa aux con 0x00.
        clrf   cont,0           ;Se inicializa el contador de bits en 0x00.
        clrf   cocint,0         ;Se inicializa el cociente en 0x00.
        movlw  0x00             ;Se compara el divisor con 0 para evitar
        cpfseq divsr            ;realizar la división/0.
        goto   VERIFICA         ;Si no es 0 se salta a VERIFICA.
        clrf   cocint,0         ;Si el divisor es 0 se retorna cociente.
        clrf   resid,0          ;y residuo = .0.
        return
```

```
VERIFICA:
        movf divdnd,W       ;En VERIFICA se analiza si el dividendo.
        cpfsgt divsr        ;es mayor o igual al divisor.
        goto  DIVISION      ;Si es así, se salta a DIVISION.
        clrf  cocint,0      ;En caso contrario se retorna con el cociente
        clrf  resid,0       ;y el residuo = .0.
        return
DIVISION:                   ;Proceso de DIVISION.
        rlcf divdnd,f       ;Se rota un bit a la izquierda el dividendo.
        rlcf aux,f          ;El Carry se carga en aux.
        movf divsr,W        ;Se carga el divisor en W.
        cpfslt aux,0        ;Se compara aux con el divisor. Si aux< divisor no
                            ;hay cambios.
        call   RESTA        ;Si aux es >= divisor hay que restar el divisor.
        rlncf cocint,f      ;Se rota el cociente 1 bit a la izquierda.
        incf   cont,f,0     ;Se incrementa el contador.
        movlw  .8           ;Si el contador es .8 entonces se terminó la división.
        cpfseq cont         ;Si cont = .8 salta una instrucción.
        goto   DIVISION     ;En caso que cont<8 se continua operando.
        movf aux,W          ;El valor restante en aux es el residuo.
        movwf resid         ;Se guarda el residuo en el registro resid.
        rrncf cocint,f      ;Se compensa el corrimiento a la izquierda demás.
        return              ;Se desplaza una a la derecha y se retorna.

RESTA:
        bsf  cocint,0       ;Si hay resta el bit q = 1.
        movf divsr,W        ;Se resta el divisor de aux.
        subwf aux,f         ;El resultado se guarda en aux y se retorna.
        return
        END
```

Ejemplo 5.7

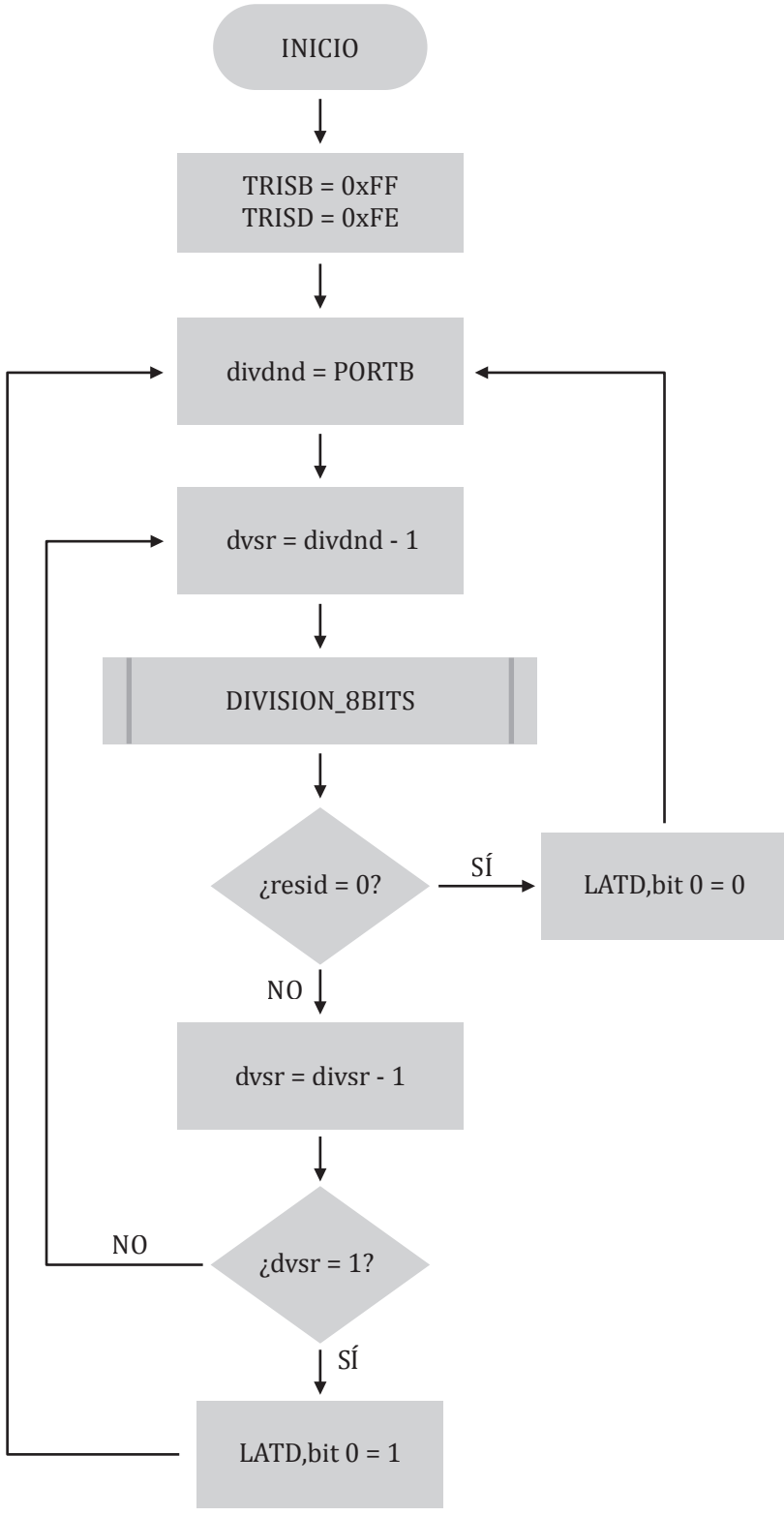
Una vez que se tiene el procedimiento para dividir cifras de 8 bits, es posible diseñar una rutina detectora de números primos. Un número primo es aquel que solo es divisible por sí mismo o por la unidad. Por tal motivo, la rutina DIVISION_8BITS debe ser usada para esta aplicación.

Si se ingresa un número binario de 8 bits por el Puerto B y la cifra es prima, entonces el pin RD0 se pone a '1'. En caso contrario, se mantendrá en '0'. El procedimiento se muestra en el diagrama de flujo del gráfico 5.14. El algoritmo se basa en realizar divisiones sucesivas del dividendo usando un divisor que parte del valor del dividendo menos 1 hasta llegar a 2. Si durante todas estas operaciones de división el residuo siempre es diferente de cero, el número es primo. Si alguna vez el residuo es 0, entonces el número no será primo y el pin RD0 se pondrá en nivel '1'. En el programa 5.11 se muestra el procedimiento en lenguaje ensamblador.

Programa 5.11. Algoritmo detector de números primos

```
#include <P18F4550.INC>
        org 0x00
        goto MAIN
        org 0x08
        retfie
        org 0x18
        retfie
        org 0x200
MAIN:
        movlw 0xFE
        movwf TRISD             ;El pin LSB del Puerto D es salida digital
        setf  TRISB             ;Todos los pines del Puerto B son entradas digitales
INICIO:
        movf  PORTB,W           ;W = Puerto B
        movwf divdnd            ;divdnd = Puerto B
        movwf divsr             ;divsr = divdnd
VERIFICACION:
        decf  divsr,f           ;divsr = divsr - 1
        call  DIVISION_8BITS    ;Se invoca a la rutina de división de 8 bits
        movlw .0                ;W=0
        cpfseq resid            ;Si el residuo = 0 salta
        goto  SIGUE             ;El residuo no es 0, por tanto salta a SIGUE
        bcf   LATD,0            ;El residuo = 0, por lo tanto, el número no es primo.
                                ;Bit 0 del Puerto D = 0
        goto  INICIO            ;Se inicia nuevamente el proceso de análisis
SIGUE:
        movlw .2                ;W = .2
        cpfseq divsr            ;Se compara el divisor con .2
        goto  VERIFICACION      ;Si es diferente, entonces se continua dividiendo
        bsf   LATD,0            ;Si es igual a .2 y se llegó a este punto el número tiene
                                ;que ser primo
        goto  INICIO            ;Se repite el proceso
```

Gráfico 5.14. Diagrama de flujo del algoritmo detector de números primos



Elaboración propia.

5.5 Detección de teclas pulsadas en un teclado matricial

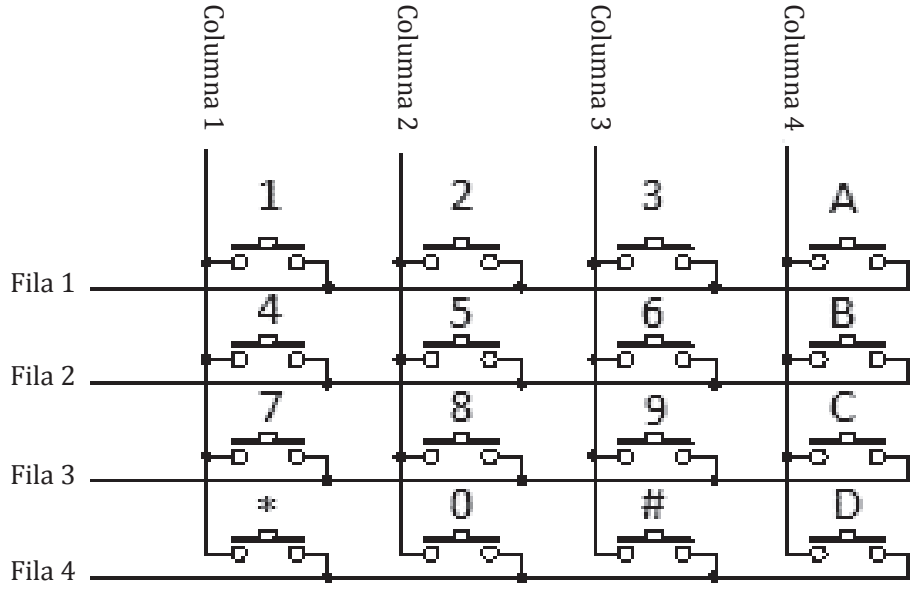
El teclado matricial es un dispositivo periférico que contiene teclas en un arreglo de filas y columnas. Cada tecla permite que el usuario pueda ingresar información al sistema embebido de tal manera que pueda interactuar con el programa residente en su memoria. Este tipo de teclado existe en diversos tamaños y formatos, es bastante utilizado en una serie de sistemas que se pueden encontrar en el mercado local. Por ejemplo, se hallan en balanzas digitales, cajas fuertes, alarmas para casas, expendedores de combustible, cajeros automáticos, etc. En el gráfico 5.15 se observa una imagen de un teclado matricial de cuatro filas y cuatro columnas.

Gráfico 5.15. Teclado matricial de 4 filas y 4 columnas

Elaboración propia.

Un teclado matricial posee un conector de ocho pines. Cuatro de estos pines representan a las filas y los otros cuatro a las columnas. Cada tecla es un pulsador que al ser pulsado genera un cortocircuito entre una columna y una fila. En el gráfico 5.16 se observa el circuito esquemático de un teclado matricial. La pulsación de cualquiera de las 16 teclas crea continuidad entre una fila y una columna específica para esa tecla. De esta manera, es posible detectar cuál fue la posición de la tecla pulsada.

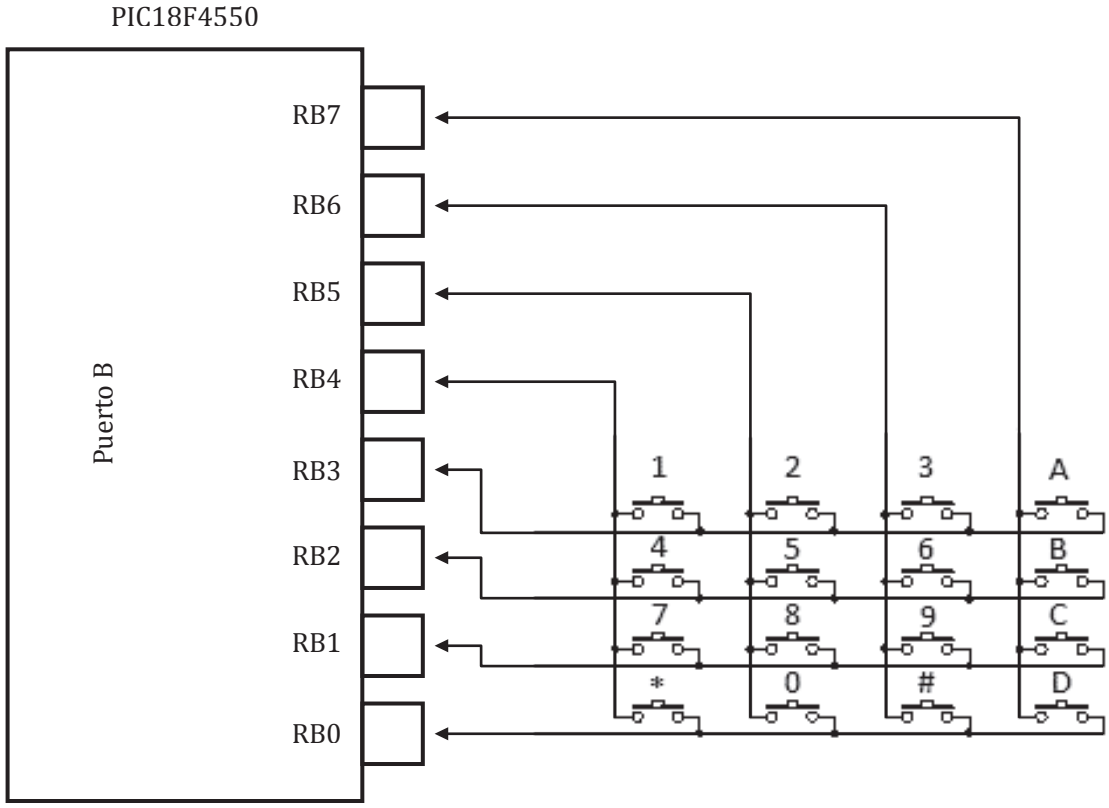
Gráfico 5.16. Diagrama descriptivo de un teclado matricial de 4 filas y 4 columnas



Elaboración propia.

Para la detección de la tecla pulsada es posible conectar directamente las filas y columnas del teclado al Puerto B del microcontrolador PIC18F4550, tal como se muestra en el diagrama del gráfico 5.17. Se elige el Puerto B del microcontrolador por ser el que cuenta con resistencias de *Pull Up* internas, que son activadas o desactivadas de los 8 pines mediante el bit RBPU del registro de función especial INTCON2 (gráfico 5.18). La ventaja de poder activar las resistencias de *Pull Up* es que al hacer una lectura del Puerto B (previamente configurado como entrada digital) permitirá detectar un estado lógico de '1' si es que todos los pines se encuentran sin conexión. Esta característica es sumamente importante para la detección de teclas pulsadas, ya que la mayor parte del tiempo las teclas se encontrarán abiertas. Cabe resaltar que la propiedad de activación de las resistencias de *Pull Up* internas se aplica a todos los pines del Puerto B. O se activan todos o se desactivan todos mediante el bit RBPU del registro INTCON2. El *Pull Up* interno no afecta la función del pin cuando este se configura como salida digital.

Gráfico 5.17. Diagrama de conexiones del teclado matricial al microcontrolador

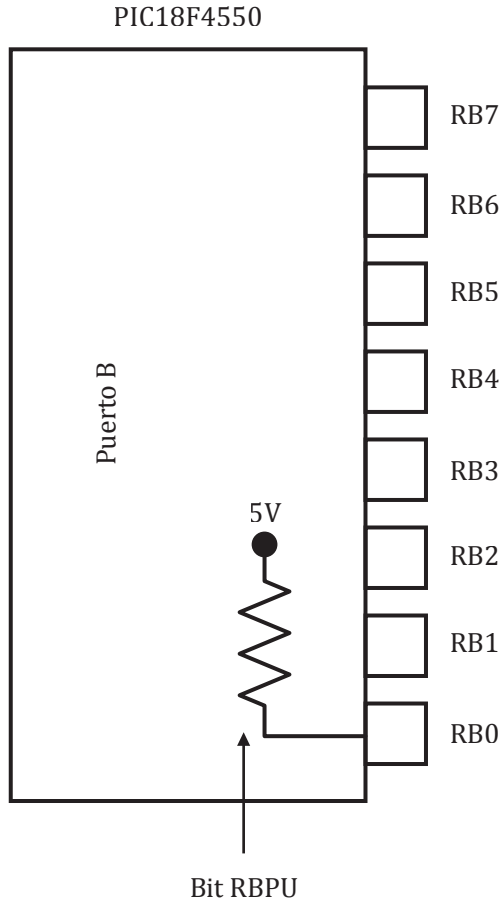


Elaboración propia.

El procedimiento para detectar una tecla pulsada es el siguiente:

Paso 1: activar las resistencias de *Pull Up* interno del Puerto B.
Paso 2: configurar los cuatro pines más significativos del Puerto B como salidas digitales y colocar cada pin a nivel '0', tal como se muestra en el gráfico 5.19.
Paso 3: configurar los cuatro pines menos significativos como entrada digital. Esto mantendrá los niveles lógicos en estos cuatro pines en estado alto cuando los pulsadores se encuentren levantados.
Paso 4: esperar a que se detecte un nivel '0' en uno de los pines de entrada. El pin que detecte el nivel bajo permitirá conocer el número de fila a la cual pertenece la tecla pulsada.
Paso 5: inmediatamente después, configurar los cuatro pines más significativos del Puerto B como entradas digitales. Esto mantendrá sus niveles lógicos en estado alto en aquellos pines que correspondan a las columnas que contienen a los pulsadores levantados.

Gráfico 5.18. Resistencias de *Pull Up* interno del Puerto B del PIC18F4550



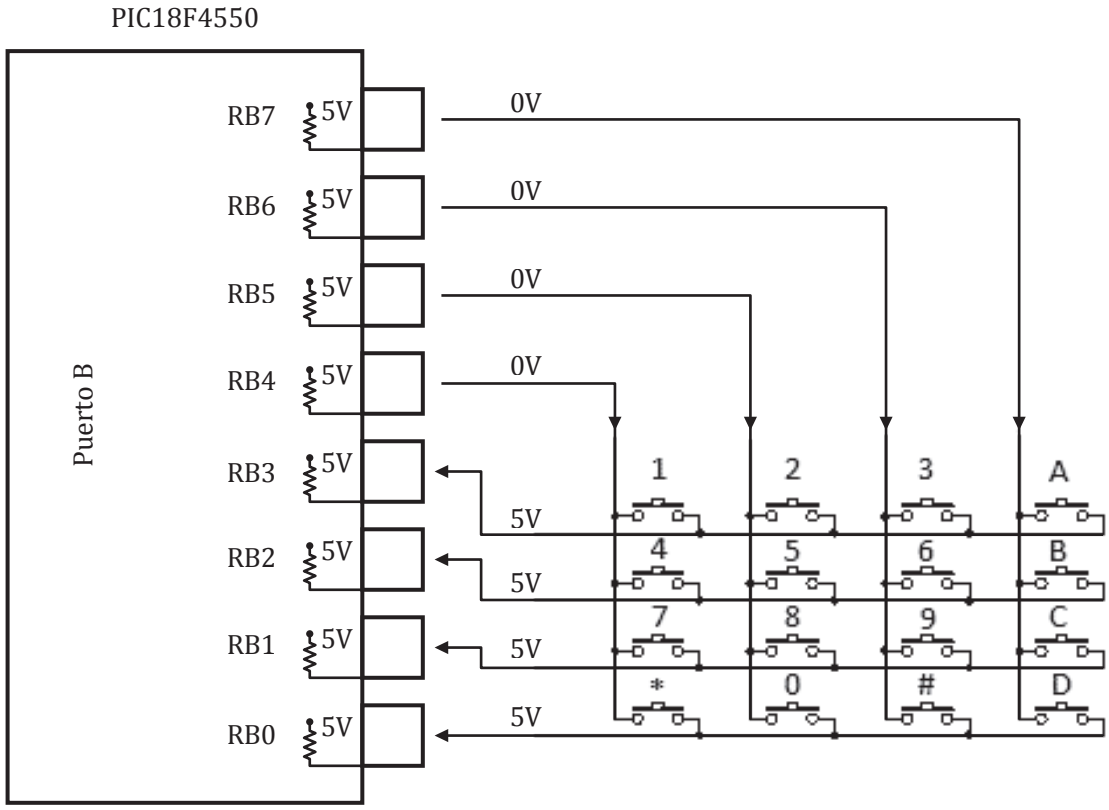
Elaboración propia.

Paso 6: configurar los cuatro pines menos significativos del Puerto B como salidas digitales con nivel '0'.
Paso 7: dado que los pasos 5 y 6 se hacen en pocos microsegundos y la tecla pulsada es generada por acción mecánica del dedo sobre el pulsador, la tecla seguirá haciendo el cortocircuito después del cambio de estado de los pines del puerto, por lo cual uno de los pines más significativos del Puerto B detectará el '0', permitiendo así detectar el valor de la columna involucrada.
Paso 8: conociendo la fila y la columna responsables del cortocircuito, se detecta la tecla pulsada. Se espera un tiempo de alrededor de 100 milisegundos para ignorar las perturbaciones producidas por el rebote de la tecla (*Glitch*) y se vuelve a configurar el Puerto B a su estado inicial dado en el paso 2.

Por ejemplo, en el diagrama del gráfico 5.20, suponiendo que se pulsa la tecla con el membrete "0" del teclado matricial. Esta tecla pertenece a la intersección Fila 4 – Columna 2, por lo tanto, el pin RB0, conectado a la fila 4, tomará el valor de '0'.

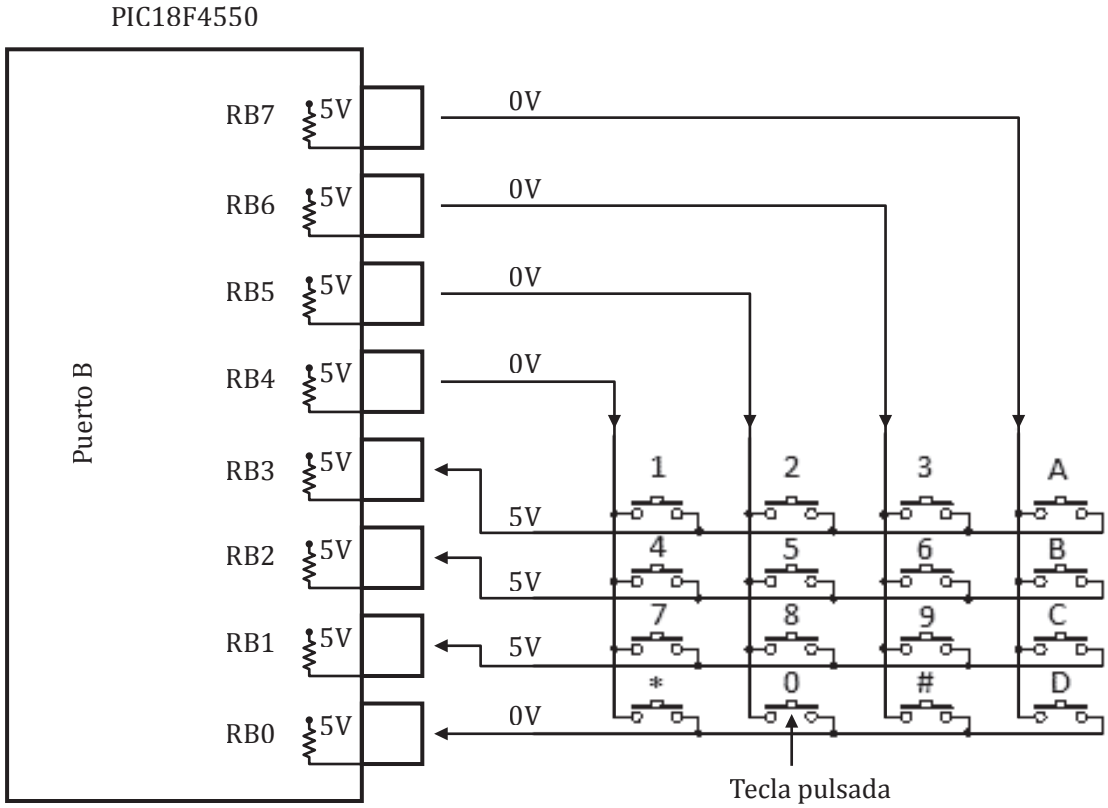
El programa detectará el nivel '0' lógico en el pin RB0, con lo cual se descifrará que la tecla pulsada se encuentra en la fila 4. Inmediatamente, se deberá hallar la columna responsable. Para esto, los pines RB3-RB0 se configuran como salida digital a nivel '0' lógico y los pines RB7-RB4 se configuran como entrada digital con el *Pull Up* activo. Una vez realizado el cambio, el nivel '0' lógico ubicado en el pin RB0 estará presente en el pin RB5 (columna 2), con lo cual el programa, al detectar el nivel '0' lógico en este pin, reconocerá la columna 2 como la responsable. Este esquema se muestra en el gráfico 5.21.

Gráfico 5.19. Configuración inicial del Puerto B para la detección de teclas pulsadas



Elaboración propia.

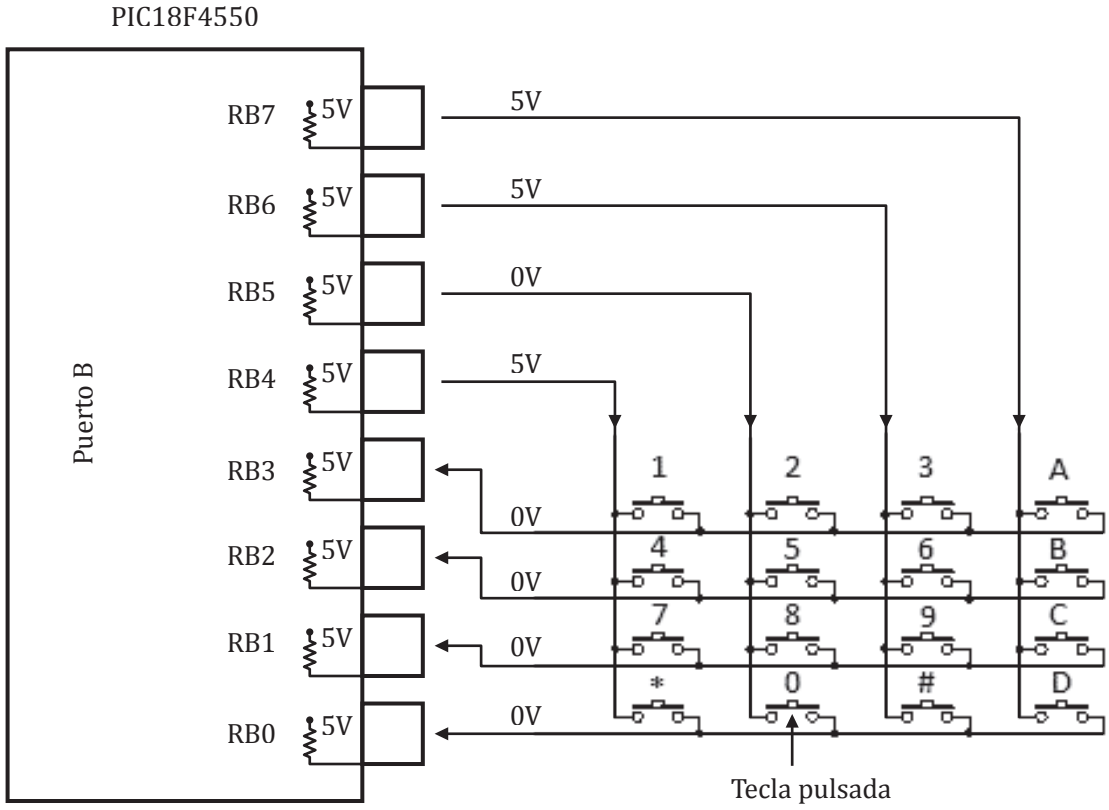
Gráfico 5.20. Ejemplo de pulsación de la tecla '0' del teclado matricial



Elaboración propia.

Una vez detectada la tecla pulsada, se deberá esperar un tiempo entre 100 y 200 milisegundos para volver a configurar el Puerto B en su modo inicial y detectar la siguiente tecla pulsada. Este retardo se aplica para ignorar la interferencia producida por el rebote mecánico de la tecla originado al ser pulsada mediante acción mecánica. En el programa 5.13 se muestra un código de ejemplo que detecta las teclas pulsadas de un teclado matricial conectado al Puerto B, con la diferencia de que las columnas se conectan a los pines menos significativos y las filas a los más significativos.

Gráfico 5.21. Cambio de configuración de los pines del Puerto B, detección de la columna



Elaboración propia.

Programa 5.13. Rutina de detección de teclas pulsadas en un teclado matricial

```
#include <P18F4550.INC>
        cblock 0x0000
                aux1
                aux2
                aux3
        endc
        ORG 0x0000
        goto MAIN
        ORG 0x0020
MAIN:
        clrf  TRISD             ;Todos los pines del Puerto D son salidas digitales
                                ;(conexión a los LEDS).
        movlw 0xF0              ;Los pines RB7-RB4 son entradas (Col1-Col4) y los pines.
        movwf TRISB,0           ;RB3-RB0 son salidas (Fil1-Fil4).
        clrf  INTCON2           ;Se activa el pull up interno del Puerto B.
        clrf  LATB              ;Se ponen los pines RB3-RB4 en '0' lógico.
```

```
FIN:
        call  LEE_TECLA         ;Se realiza la lectura de una tecla pulsada.
        movlw 0xF0              ;Se configura nuevamente los pines RB7-RB4 como
        movwf TRISB,0           ;entradas y RB3-RB0 salidas.
        clrf  LATB              ;Se ponen los pines RB3-RB0 en '0' lógico.
        call  DELAY200MS        ;Se realiza un retardo de 200ms para obviar los glitches.
        goto FIN                ;Se repite la secuencia.
LEE_TECLA:
        btfss PORTB,7,0         ;Si RB7 = '1' no hay tecla pulsada en la columna 1.
        bra   COLUMNA1          ;Si RB7 = '0' hay una tecla pulsada en la columna 1.
        btfss PORTB,6,0         ;Si RB6 = '1' no hay tecla pulsada en la columna 2.
        bra   COLUMNA2          ;Si RB6 = '0' hay una tecla pulsada en la columna 2.
        btfss PORTB,5,0         ;Si RB5 = '1' no hay tecla pulsada en la columna 3.
        bra   COLUMNA3          ;Si RB5 = '0' hay una tecla pulsada en la columna 3.
        btfss PORTB,4,0         ;Si RB4 = '1' no hay tecla pulsada en la columna 4.
        bra   COLUMNA4          ;Si RB4 = '0' hay una tecla pulsada en la columna 4.
        return
COLUMNA1:                       ;Se sabe que la tecla está en la columna 1, hay que
                                ;averiguar la fila.
        movlw 0x0F              ;Se cambia el estado del Puerto B. Ahora RB7-RB4 son
      movwf TRISB,0             ;salidas y RB3-RB0 son entradas.
        clrf  LATB              ;Se ponen RB7-RB4 a '0' lógico.
        btfss PORTB,3,0         ;Si RB3 = '1' no hay tecla pulsada en la fila 1.
        movlw .1                ;Si RB3 = '0' hay tecla pulsada en la fila 1.
        btfss PORTB,2,0         ;Si RB2 = '1' no hay tecla pulsada en la fila 2.
        movlw .5                ;Si RB2 = '0' hay tecla pulsada en la fila 2.
        btfss PORTB,1,0         ;Si RB1 = '1' no hay tecla pulsada en la fila 3.
        movlw .9                ;Si RB1 = '0' hay tecla pulsada en la fila 3.
        btfss PORTB,0,0         ;Si RB0 = '1' no hay tecla pulsada en la fila 4.
        movlw .13               ;Si RB0 = '0' hay tecla pulsada en la fila 4.
        movwf LATD,0            ;Se muestra el código binario respectivo en los LEDs del
                                ;Puerto D.
        return                  ;Se retorna de la rutina de lectura de teclado
COLUMNA2:                       ;Se sabe que la tecla está en la columna 2, se debe
                                ;averiguar la fila.
        movlw 0x0F              ;Se cambia el estado del Puerto B. Ahora RB7-RB4 son
                                ;salidas.
        movwf TRISB,0           ;y RB3-RB0 son entradas.
        clrf  LATB              ;Se ponen RB7-RB4 a '0' lógico.
        btfss PORTB,3,0         ;Si RB3 = '1' no hay tecla pulsada en la fila 1.
        movlw .2                ;Si RB3 = '0' hay tecla pulsada en la fila 1
```

```
        btfss PORTB,2,0     ;Si RB2 = '1' no hay tecla pulsada en la fila 2
        movlw .6            ;Si RB2 = '0' hay tecla pulsada en la fila 2
        btfss PORTB,1,0     ;Si RB1 = '1' no hay tecla pulsada en la fila 3
        movlw .10           ;Si RB1 = '0' hay tecla pulsada en la fila 3
        btfss PORTB,0,0     ;Si RB0 = '1' no hay tecla pulsada en la fila 4
        movlw .14           ;Si RB0 = '0' hay tecla pulsada en la fila 4
        movwf LATD,0        ;Se muestra el código binario respectivo en los LEDs del
                            ;Puerto D.
        return              ;Se retorna de la rutina de lectura de teclado.
COLUMNA3:                   ;Se sabe que la tecla esta en la columna 3, hay que
                            ;averiguar la fila.
        movlw 0x0F          ;Se cambia el estado del Puerto B. Ahora RB7-RB4 son
        movwf TRISB,0       ;salidas y RB3-RB0 son entradas.
        clrf  LATB          ;Se ponen RB7-RB4 a '0' lógico.
        btfss PORTB,3,0     ;Si RB3 = '1' no hay tecla pulsada en la fila 1.
        movlw .3            ;Si RB3 = '0' hay tecla pulsada en la fila 1.
        btfss PORTB,2,0     ;Si RB2 = '1' no hay tecla pulsada en la fila 2.
        movlw .7            ;Si RB2 = '0' hay tecla pulsada en la fila 2.
        btfss PORTB,1,0     ;Si RB1 = '1' no hay tecla pulsada en la fila 3.
        movlw .11           ;Si RB1 = '0' hay tecla pulsada en la fila 3.
        btfss PORTB,0,0     ;Si RB0 = '1' no hay tecla pulsada en la fila 4.
        movlw .15           ;Si RB0 = '0' hay tecla pulsada en la fila 4.
        movwf LATD,0        ;Se muestra el código binario respectivo en los LEDs del
        return              ; Puerto D. Se retorna de la rutina de lectura de teclado
COLUMNA4:                   ;Se sabe que la tecla está en la columna 4, hay que
                            ;averiguar la fila
        movlw 0x0F          ;Se cambia el estado del Puerto B. Ahora RB7-RB4 son
                            ;salidas.
        movwf TRISB,0       ;y RB3-RB0 son entradas.
        clrf  LATB          ;Se ponen RB7-RB4 a '0' lógico.
        btfss PORTB,3,0     ;Si RB3 = '1' no hay tecla pulsada en la fila 1.
        movlw .4            ;Si RB3 = '0' hay tecla pulsada en la fila 1.
        btfss PORTB,2,0     ;Si RB2 = '1' no hay tecla pulsada en la fila 2.
        movlw .8            ;Si RB2 = '0' hay tecla pulsada en la fila 2.
        btfss PORTB,1,0     ;Si RB1 = '1' no hay tecla pulsada en la fila 3.
        movlw .12           ;Si RB1 = '0' hay tecla pulsada en la fila 3.
        btfss PORTB,0,0     ;Si RB0 = '1' no hay tecla pulsada en la fila 4.
        movlw .16           ;Si RB0 = '0' hay tecla pulsada en la fila 4.
        movwf LATD,0        ;Se muestra el código binario respectivo en los LEDs del
                            ;Puerto D.
        return              ;Se retorna de la rutina de lectura de teclado
```

```
DELAY200MS:
        movlw .3
        movwf aux1,0
RET1:
        movlw .110
        movwf aux2,0
RET2:
        movlw .201
        movwf aux3,0
RET3:
        decfsz aux3,f,0
        goto   RET3
        decfsz aux2,f,0
        goto   RET2
        decfsz aux1,f,0
        goto   RET1
        return
        END
```

Video

Operaciones aritméticas con el lenguaje ensamblador del PIC18F4550.

http://bit.ly/1UCHzuE

Resolución de videos en HD

Preguntas de repaso

1. Analice el siguiente programa basado en la conexión del PIC18F4550 y en el Display 7 segmentos ánodo común del gráfico 5.5:

```
        cblock 0x20
                aux
                aux1
                aux2
                aux3
        endc
        org 0x00
        goto MAIN
        org 0x20
MENSAJE: db 0x41,0x0C,0x46,0x3F,0x0C,0x4F,0x46,0x00
        org 0x200
MAIN:
        movlw 0x80
        movwf TRISD
        movlw UPPER MENSAJE
        movwf TBLPTRU
        movlw HIGH MENSAJE
        movwf TBLPTRH
        movlw LOW MENSAJE
        movwf TBLPTRL
INICIO:
        TBLRD*+
        movf  TABLAT,W
        movwf aux
        movlw 0x00
        xorwf aux,W
        btfsc STATUS,Z
        goto  MAIN
        movff aux,LATD
        call  RETARDO500MS
        goto  INICIO
```

¿Cuál es el mensaje que se visualiza en los Displays 7 segmentos?

2. Diseñe un programa para que el Display 7 segmentos ánodo común del gráfico 5.6 muestre el mensaje «Hola Mundo» generando cada carácter por 0.5 segundos.
3. Explique cuál es la función de las resistencias de *Pull Up* en la rutina de detección de teclas pulsadas del teclado matricial. ¿Qué pasaría si estas resistencias no se activaran?
4. Diseñe una rutina que determine si un número es divisible entre 5.
5. Partiendo del circuito del gráfico 5.4, diseñe un programa que lea un valor binario de 8 bits del Puerto B, realice la conversión a BCD y muestre cada cifra durante 1 segundo en el Display 7 segmentos.

Respuestas

Respuestas del capítulo 5:

http://bit.ly/1EMNoUT

Capítulo 6. Interrupciones

Hasta el momento se han diseñado programas en lenguaje ensamblador que ejecutan instrucciones en un orden secuencial determinado para un proceso en particular. Sin embargo, un microcontrolador es un dispositivo multitarea que debe atender a diversos procesos casi en simultáneo, lo cual no es una misión simple. Muchas veces, algunos de estos procesos no requieren atención completa, sino que deben ser ejecutados bajo ciertas condiciones externas que suelen ser poco frecuentes. Por esta razón, si el procesador dedica su atención completa a todos estos eventos, se tendrá, aparte de un serio problema de sincronismo de tareas, un alto e innecesario consumo de energía.

Para evitar estos inconvenientes, el PIC18F cuenta con un sistema de interrupciones que permite distraer al CPU de determinada tarea cuando cierto evento externo ha ocurrido. Este mecanismo es fundamental para que el sistema embebido pueda operar de manera eficiente atendiendo múltiples procesos y a su vez manteniendo un consumo de energía mínimo.

El presente capítulo trata sobre el concepto de una interrupción y los tipos de interrupción existentes en el PIC18F. Además, muestra la forma de configuración y ejemplos aplicados para el manejo de las interrupciones más sencillas que posee el microcontrolador.

6.1 El concepto de interrupción

Suponiendo que un programador se encuentra programando su microcontrolador en casa, sentado en un escritorio, utilizando una computadora portátil. Su principal tarea es diseñar un algoritmo para implementar una alarma de seguridad, y debe terminar esta tarea pronto, ya que el prototipo del *firmware* debe estar finalizado al día siguiente para su análisis por parte de los ingenieros de desarrollo de su empresa. Sin embargo, está aguardando una llamada telefónica importante y sabe que esta puede llegar en cualquier momento del día. De la misma manera, espera una visita importante también, que va llegar a su casa y a tocarle la puerta en cualquier momento del día.

Tanto la llamada telefónica como la visita son fundamentales para él, y no puede perderlas. Se sabe que la llamada telefónica va a timbrar su teléfono por 30 segundos y, si no contesta, perderá la llamada. Por otro lado, la visita tocará la puerta por un minuto y, de no recibir respuesta, optará por retirarse.

El teléfono se encuentra en la sala, en un ambiente alejado de su escritorio, y la puerta de su hogar también. El problema es que tanto el teléfono como el timbre de la puerta se encuentran dañados, por lo tanto, no tiene cómo enterarse de si se presenta una llamada telefónica entrante o una persona en la puerta. La única forma para no perder estos eventos es que se acerque al teléfono al menos una vez cada 30 segundos y verifique que no hay una llamada entrante, mirando la pantalla LCD alfanumérica del equipo. De la misma manera, al menos una vez cada minuto, deberá abrir la puerta de su casa para verificar que no haya alguna persona intentando tocar el timbre.

Se pasa mirando el teléfono dos veces por minuto y abriendo la puerta cada minuto. La gran mayoría de veces no hay llamadas entrantes ni personas en la puerta. Esto lo obliga a desatender su tarea principal: el desarrollo del *firmware* de la alarma, con lo cual su trabajo se ve afectado. Por otro lado, se desgasta, ya que el hecho de levantarse de su escritorio continuamente lo mantiene agotado.

El problema tendría solución si el timbre de la puerta estuviera reparado y la bocina del teléfono estuviera en buen estado. De esta manera, el programador solo se levantaría a atender el teléfono y la puerta una sola vez y esto se haría únicamente en el momento en que sea necesario, de manera que no desperdiciaría el tiempo y podría terminar la tarea principal: el *firmware* de la alarma. Adicionalmente, el programador no estaría tan cansado por tener que levantarse de la silla constantemente cada minuto.

En el ejemplo presentado, el timbre de la puerta y la bocina del teléfono pueden considerarse interrupciones, ya que interrumpen al programador de su tarea principal para que se dedique a realizar otro proceso diferente. Sin embargo, la interrupción solo ocurre en el momento apropiado, de manera que el programador solo asistirá a abrir la puerta o a contestar el teléfono cuando sea necesario.

Otro concepto importante que se debe tomar en cuenta es la prioridad de la interrupción. ¿Qué pasaría si la bocina del teléfono y el timbre de la puerta sonaran al mismo tiempo? Ante esta situación, el programador deberá definir qué es más importante para él: o contestar el teléfono primero o abrir la puerta. Pues bien, esto es un aspecto que no puede evitarse y el programador ha de tener en claro desde un comienzo cuál es el evento prioritario.

Entrando al tema de los sistemas embebidos, una interrupción es un evento que puede modificar el flujo natural de un programa y desviarlo a otra dirección, donde se ejecutará otro subprograma, para, finalmente, retornar al punto de ejecución donde se había quedado el procesador antes de la interrupción. La interrupción puede originarse en cualquier momento, pero siempre se retornará al punto donde el puntero de programa se encontraba antes de que ocurriera la interrupción. Una interrupción opera de manera similar a la instrucción *call* <etiqueta>, con la diferencia de que esta instrucción se ejecuta siempre en la misma dirección de la memoria de programa. Véase el siguiente ejemplo:

```
MAIN:
        call TAREA1
        call TAREA2
        call TAREA3
goto MAIN
```

Suponiendo que la TAREA2 contiene un procedimiento que solo se ejecuta si se ha recibido un pulso externo por uno de los pines de interrupción externa; de otra manera, simplemente retorna. La TAREA3, por otro lado, ejecuta otro procedimiento que depende de la pulsación de una tecla en un teclado matricial externo. De no haber ocurrido este evento, simplemente se retorna de esta rutina. Esto implica que el programa principal solamente ejecuta las instrucciones de la TAREA1, mientras que la mayor parte del tiempo las otras dos tareas simplemente no requieren ser atendidas, lo cual genera un gasto de energía y tiempo al procesador.

Lo ideal es que la rutina que ejecuta el procesador se transforme en lo siguiente:

```
MAIN:
        call TAREA1
goto MAIN
```

Las tareas 2 y 3 no aparecen en este esquema. Sin embargo, más adelante se verá que al, ocurrir una interrupción, estas tareas podrán ser ejecutadas siendo invocadas desde una dirección específica conocida como Vector de interrupción.

El PIC18F4550 cuenta con 21 fuentes de interrupción, las cuales se dividen en interrupciones generales e interrupciones de periféricos. Entre las interrupciones generales se tienen 5 tipos diferentes (que se muestran en el gráfico 6.1). Las interrupciones de periféricos se presentan en el gráfico 6.2.

Gráfico 6.1. Lista de interrupciones generales del PIC18F4550

Interrupción
Interrupción del temporizador *Timer* 0
Interrupción por cambio de estado del Puerto B
Interrupción externa 0
Interrupción externa 1
Interrupción externa 2

Elaboración propia.

Gráfico 6.2. Lista de interrupciones de periféricos del PIC18F4550

Interrupción
Interrupción del temporizador *Timer 1*
Interrupción del temporizador *Timer 2*
Interrupción del temporizador *Timer 3*
Interrupción del módulo de comparación y captura CCP1
Interrupción del módulo de comparación y captura CCP2
Interrupción del convertidor analógico digital
Interrupción de transmisión del EUSART
Interrupción por recepción del EUSART
Interrupción del módulo MSSP
Interrupción por colisión del módulo MSSP
Interrupción del módulo comparador
Interrupción por el USB
Interrupción por falla del oscilador
Interrupción por el evento de escritura de la memoria FLASH o EEPROM
Interrupción por bajo nivel de voltaje de alimentación del PIC18F
Interrupción del SPP (*Streaming Parallel Port*)

Elaboración propia.

En el PIC18F, todas las interrupciones permanecen desactivadas por defecto. Existen interrupciones enmascarables y no enmascarables. Las primeras son las mostradas en los gráficos 6.1 y 6.2, ya que todas pueden desactivarse. Dentro de las interrupciones no enmascarables se encuentra el pulso de Reset o una falla por *software*, que siempre generarán el reinicio del programa, lo cual se puede considerar como una interrupción no enmascarable o no inactiva.

Las interrupciones generales del PIC18F se habilitan mediante el bit GIE (del inglés *Global Interrupt Enable*) del registro INTCON que se muestra en el gráfico 6.3. Las interrupciones de periféricos se habilitan, además, con el bit PEIE (del inglés *Peripheral Interrupt Enable*) en '1'. El bit GIE se conoce como la máscara global de interrupciones y el bit PEIE como la máscara de interrupciones de periféricos del PIC18F. Estos bits también cumplen con otra función adicional dependiendo de la configuración del bit IPEN (del inglés *Interrupt Priority Enable*) del registro RCON. El bit GIE se transforma en GIEH si IPEN se configura en nivel '1' y permite habilitar las interrupciones de alta prioridad. Así también, el bit PEIE se transforma en GIEL, el cual permite habilitar las interrupciones de baja prioridad.

Los bits TMR0IE, INT0IE y RBIE son los bits habilitadores de las interrupciones individuales del Temporizador *Timer 0*, Interrupción externa 0 y cambio de estado del Puerto B, respectivamente. Existen otros registros con bits habilitadores para el resto de interrupciones, como se verá más adelante. Colocando cada uno de estos bits de manera individual a nivel '1', se activa cada interrupción. La interrupción puede ser desactivada simplemente colocando cada uno de estos bits a nivel '0'.

Gráfico 6.3. Bits del registro INTCON

GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

Elaboración propia.

Los bits TMR0IF, INT0IF y RBIF se conocen como las banderas de interrupción o banderas de estado. Cuando uno de estos bits se pone a '1' de manera automática es debido a que el evento que dispara la interrupción ha ocurrido. Por ejemplo, en el TMR0, cada vez que el conteo pasa de 0xFF a 0x00 se produce un evento de desbordamiento, lo cual genera que el bit TMR0IF se ponga a nivel '1' y la interrupción por el TMR0 se active. Las banderas de interrupción permiten identificar cuál fue el evento que causó el proceso de interrupción, para así determinar qué proceso o rutina de servicio de interrupción ejecutar.

Cada módulo del microcontrolador que cuenta con una fuente de interrupción posee tres bits de configuración: el bit habilitador, el bit de bandera y el bit de selección de interrupción de alta o baja prioridad.

En general existen 10 registros de función especial que permiten configurar las 21 interrupciones del microcontrolador junto con otras funciones generales. Estos registros son los siguientes:

1. INTCON
2. INTCON2
3. INTCON3
4. RCON

5. PIR1
6. PIR2
7. PIE1
8. PIE2
9. IPR1
10. IPR2

6.2 El vector de interrupción

Un vector es una dirección especial de la memoria de programa destinada para ser apuntada por el PC en determinado evento. Por ejemplo, el vector de Reset es la dirección 0x0000 de la memoria de programa y la instrucción guardada en esta dirección siempre será ejecutada ante un evento de Reset, lo cual implica que siempre será la primera instrucción en ejecutarse. De la misma manera, un vector de interrupción es una dirección de la memoria de programa a la cual el PC apuntará siempre que ocurra un evento de interrupción.

El PIC18F contiene dos vectores de interrupción fijos en las direcciones de la memoria de programa: 0x0008 y 0x0018. El primer vector se conoce como de alta prioridad, mientras que el segundo vector se conoce como de baja prioridad.

Cada vez que ocurre un evento de interrupción, si este ha sido configurado como alta prioridad, el PC tomará el valor de 0x0008 y ejecutará la instrucción ahí almacenada. Si la interrupción es de baja prioridad, entonces el PC tomará el valor 0x0018 y ejecutará la instrucción almacenada en esta dirección (previamente, el valor anterior a la interrupción del PC será almacenado en la pila).

Retomando el ejemplo del subcapítulo anterior, en el que se tenían tres tareas, de las cuales la TAREA 1 se ejecuta constantemente, mientras que la Tarea 2 depende de la interrupción externa y la TAREA 3 depende del cambio de estado del Puerto B, el modelo del programa quedaría tal como se muestra en el esquema del programa 6.1.

Programa 6.1. Esquema de operación en base a dos interrupciones

```
      org 0x00
      goto MAIN
      org 0x0008              ;Dirección del vector de interrupción de alta prioridad.
      goto RUT_INT            ;Salto a la rutina de servicio de interrupción.
       org 0x0018             ;Dirección del vector de interrupción de baja prioridad.
      retfie                  ;Instrucción de retorno de la interrupción.
 org 0x0020                   ;Dirección de inicio del programa.
MAIN:
      bsf INTCON,TMR0IE       ;Se activa la interrupción por desbordamiento del TMR0.
      bsf INTCON,RBIE         ;Se activa la interrupción por cambio de estado del
                              ;Puerto B.
      bsf INTCON,GIE          ;Se habilita la máscara global de interrupciones.
```

```
INICIO:
    call  TAREA1              ;Se ejecuta la rutina TAREA1 de manera constante.
    goto INICIO
RUT_INT:                      ;Si ocurre un evento de interrupción ya sea por el cambio
    btfsc  INTCON,TMR0IF      ;de estado del Puerto B o por desbordamiento del TMR0
                              ;el PC toma el valor 0x0008 y salta a la etiqueta
                              ;RUT_INT.
    goto   TAREA2             ;Si el bit TMR0IF='1' entonces el TMR0 fue la causa de
                              ;la interrupción y se salta a la rutina TAREA 2. En caso
    btfsc  INTCON,RBIF        ;sea el bit RBIF el que está en estado '1', entonces el
    goto   TAREA3             ;cambio de estado del Puerto B es el causante de la
                              ;interrupción y se salta a la rutina TAREA 3.
    retfie                    ;En caso contrario, la fuente de la interrupción fue otra
                              ;no conocida y simplemente se retorna de la interrupción.
```

Como se puede observar en el programa 6.1, se utiliza solamente el vector de interrupción de alta prioridad en la dirección 0x0008 para manejar las interrupciones. Cuando el programa se inicia, se habilitan solo dos interrupciones: la de desbordamiento del TMR0 y la de cambio de estado del Puerto B al poner los bits TMR0IE y RBIE a nivel '1'. Ambas interrupciones por defecto son de alta prioridad. Luego, al poner el bit GIE a nivel '1', la máscara de interrupción global queda activa, quedando habilitadas las dos interrupciones anteriores.

Luego, se ejecuta de forma continua la rutina TAREA1. En esta rutina se asume que se encuentra una secuencia de instrucciones que está siendo ejecutada de forma indefinida. Si se tiene un caso en el que ocurriese un desbordamiento del TMR0, la bandera de interrupción TMR0IF se pone a nivel '1', con lo cual se activa la interrupción. En ese momento, el valor del PC se almacena en la pila y luego toma el valor 0x0008 para ejecutar la instrucción que se encuentra en esa dirección (vector de interrupción de alta prioridad).

En la dirección 0x0008 se encuentra una instrucción de salto a la rutina de servicio de interrupción. Hasta ese momento poco se sabe de la interrupción, solo que ha habido un evento y que se debe determinar si fue producto del cambio de estado del Puerto B o del desbordamiento del TMR0. Para evaluar esto se salta a la dirección dada por la etiqueta RUT_INT.

En la rutina de servicio de interrupción RUT_INT se consulta primero si la causa de la interrupción fue el TMR0. Como la bandera TMR0IF es '1', la condición de evaluación del bit ejecuta la instrucción goto TAREA2, con lo cual se invoca a la rutina correspondiente a la interrupción por desbordamiento del TMR0.

La rutina TAREA2 debe incluir dos instrucciones fundamentales al finalizar. Estas instrucciones corresponden al borrado de la bandera de interrupción y el retorno desde la interrupción a la dirección del programa antes de la rutina de salto al vector 0x0008. Esto se observa en el programa 6.2.

Programa 6.2. Formato de la rutina TAREA2 del Programa 6.1

```
TAREA2:
        Instrucción 1
        Instrucción 2
        Instrucción 3
            .
            .
            .
        bcf INTCON,TMR0IF      ;Bandera de interrupción del TMR0 desactivada.
        retfie                 ;Retorno de la interrupción.
```

La penúltima instrucción coloca la bandera de interrupción de desbordamiento del TMR0 en nivel '0', lo cual determina que el proceso (TAREA2) ya fue atendido y no debe volver a ser ejecutado hasta que nuevamente se desborde el TMR0. Si la bandera TMR0IF no se pone a nivel '0' en el programa, la interrupción volverá a ser ejecutada nuevamente, ya que la bandera mantendrá el proceso activo.

La instrucción retfie representa el retorno de la interrupción. Es muy similar a la instrucción *Return*. La diferencia es que retfie hace referencia al retorno desde una rutina de interrupción. Esta instrucción genera que el último valor en la pila se vuelva a cargar en el registro PC, permitiendo que se retorne a la siguiente instrucción que se iba a ejecutar antes de que ocurriera el evento de la interrupción.

Si, en caso contrario, la fuente de interrupción hubiera sido generada por el cambio de estado del Puerto B, el bit RBIF se hubiera puesto a nivel '1' y la rutina a ejecutar hubiera sido TAREA3, la cual debe finalizar con el mismo par de instrucciones del programa 6.2. La única diferencia es que la bandera que se debe poner a nivel '0' es RBIF.

Durante todo el proceso de la interrupción, es decir, desde que se carga el valor 0x0008 en el PC hasta que se ejecuta la instrucción retfie, el bit GIE permanece en nivel '0', lo cual deshabilita a todas las interrupciones. Esto quiere decir que una rutina de interrupción de alta prioridad no puede ser interrumpida por otra interrupción. Si en el instante que se ejecuta la rutina TAREA2, hubiera ocurrido un evento de cambio de estado en el Puerto B, el bit RBIF se habría puesto a nivel '1', pero, recién una vez que se ha retornado de la interrupción, el PC volvería a cargarse con el valor 0x0008, ejecutando otra vez la rutina RUT_INT. Este modo de operación de las interrupciones con una sola prioridad es compatible con el modo de operación de la familia PIC16F de Microchip.

6.3 La prioridad de una interrupción

En el ejemplo visto en los programas 6.1 y 6.2 se puede notar que tanto la interrupción del cambio de estado del Puerto B como el desbordamiento del TMR0 tienen la misma prioridad. Esto implica que la interrupción que aparece primero se ejecuta primero. Mientras la rutina de interrupción está siendo ejecutada, nada la puede detener. Pero ¿qué pasaría si se quisiera que la rutina TAREA2 pueda ser interrumpida si es que ocurre un cambio de estado en el Puerto B?

Para resolver este requerimiento se necesitan definir las prioridades. Evidentemente, la interrupción por cambio de estado del Puerto B deberá tener una prioridad mayor que la del TMR0. Una vez fijada la interrupción del TMR0 en baja prioridad, esta podrá ser interrumpida por otra de alta prioridad. Una vez que se termine la rutina de servicio de interrupción de alta prioridad, la instrucción retfie retornaría al PC a la dirección de la rutina de servicio de interrupción de baja prioridad en donde se había quedado antes el PC.

El bit IPEN del registro RCON permite habilitar los modos de alta y baja prioridad. En el ejemplo anterior, el bit IPEN por defecto estaba en nivel '0', por lo cual todas las interrupciones estaban forzadas a operar en alta prioridad. Si el bit IPEN se pone a nivel '1', el bit GIE se transforma en GIEH permitiendo habilitar (con nivel '1') la máscara global de interrupciones de alta prioridad, mientras el bit PEIE se transforma en GIEL que permite habilitar (con nivel '1') la máscara global de interrupciones de baja prioridad.

Una vez que se han habilitado las máscaras de prioridades globales, se debe asignar individualmente la prioridad de cada interrupción. En el registro INTCON2 se encuentran los bits de configuración del nivel de prioridad tanto para el TMR0 como para la interrupción de cambio de estado del Puerto B. Estos bits son TMR0IP y RBIP, respectivamente, y se pueden ver en el gráfico 6.4.

Gráfico 6.4. Bits del registro INTCON2

$\overline{\text{RBPU}}$	INTDEG0	INTDEG1	INTDEG2	—	TMR0IP	—	RBIP

Elaboración propia.

Si el bit TMR0IP se pone a nivel '0', la interrupción del TMR0 se configura en baja prioridad, mientras que si se pone en nivel '1' se configura en alta prioridad. Lo mismo aplica para el bit RBIP y para todos los demás bits xxIP (IP proviene del inglés *Interrupt Priority*) de los otros registros de control de interrupción implicados.

En el programa 6.3 se ha cambiado la configuración inicial de las interrupciones, quedando el TMR0 en baja prioridad y la interrupción por cambio de estado en alta prioridad.

Programa 6.3. Ejemplo de interrupciones configuradas en alta y baja prioridad

```
#include <P18F4550.INC>
        org 0x00
        goto MAIN
        org 0x08
        goto TAREA3
        org 0x018
        goto TAREA2
        org 0x20
```

```
MAIN:
        bsf RCON,IPEN          ;Se activan las prioridades de las interrupciones
        bcf INTCON2,TMR0IP     ;Prioridad baja para TMR0
        bsf INTCON2,RBIP       ;Prioridad alta para el cambio de estado del Puerto B.
        bsf INTCON,RBIE        ;Se habilita la interrupción por cambio de estado del
                               ;Puerto B.
        bsf INTCON,TMR0IE      ;Se habilita la interrupción por cambio de estado del
                               ;TMR0.
        bsf INTCON,GIEL        ;Se habilita la máscara de interrupciones de baja
                               ;prioridad.
        bsf INTCON,GIEH        ;Se habilita la máscara de interrupciones de alta
                               ;prioridad.
INICIO:
        movlw '0'              ;PC=0x2E
        movwf 0x30,0           ;PC=0x30
        addlw 0x04             ;PC=0x32
        movwf 0x30,0           ;PC=0x34
        incf  0x30,f,0         ;PC=0x36
      goto INICIO              ;PC=0x38
TAREA2:
        btfss INTCON,TMR0IF    ;PC=0x3C
        retfie                 ;PC=0x3E
        nop                    ;PC=0x40
        nop                    ;PC=0x42
        nop                    ;PC=0x44
        bcf   INTCON,TMR0F     ;PC=0x46
        retfie                 ;PC=0x48
TAREA3:
        btfss INTCON,RBIF      ;PC=0x4A
        retfie                 ;PC=0x4C
        nop                    ;PC=0x4E
        nop                    ;PC=0x50
        nop                    ;PC=0x52
        bcf  INTCON,RBIF       ;PC=0x54
        retfie                 ;PC=0x56
        END
```

Como se puede observar en este ejemplo, primero se habilitan las prioridades de las interrupciones con el bit IPEN a nivel lógico ‘1’. Luego, se configura la interrupción del TMR0 en baja prioridad (con el bit TMR0IP = ‘0’) y la interrupción de cambio de estado del puerto B en alta prioridad (con el bit RBIP = ‘1’). Después se habilitan las interrupciones de manera individual y, finalmente, se activan las máscaras globales de las interrupciones de prioridad alta (GIEH = ‘1’) y prioridad baja (GIEL = ‘1’).

Posteriormente, se ejecutan las instrucciones entre las direcciones 0x2E y 0x38 de manera secuencial. Si ocurriera la interrupción por cambio de estado del Puerto B el bit RBIF valdría '1', con lo cual el PC guardaría la dirección en curso en la pila (suponiendo la dirección 0x36) y tomaría el valor 0x0008, ejecutando luego la instrucción goto TAREA3. Esta instrucción lleva al PC al valor de 0x004A, con lo cual se ejecutan las instrucciones hasta la dirección 0x56. Si durante la ejecución de estas instrucciones ocurriera una interrupción por desbordamiento del TMR0, el PC no se alteraría hasta que retorne de la interrupción a la dirección 0x36. Una vez allí, el PC guarda nuevamente el valor 0x36 en la pila y toma el valor 0x0018, con lo cual ejecuta la instrucción goto TAREA2.

Si la interrupción de cambio de estado del Puerto B ocurriera durante la ejecución de las instrucciones de la rutina TAREA2 (por ejemplo, antes de ejecutar la instrucción de la dirección 0x40), entonces el PC cargará el valor 0x08 y almacenará su valor previo en la pila. Luego, saltará a la dirección TAREA3 y, al ejecutar la instrucción retfie de esta rutina, se retornará a la última instrucción ejecutada en la rutina TAREA2 (dirección 0x40). Cuando se ejecute la instrucción retfie por segunda vez, el PC retornará a la dirección inicial 0x36 donde se había quedado.

En resumen, una interrupción de alta prioridad puede interrumpir a una de baja prioridad, pero no a otra de alta prioridad. Una interrupción de baja prioridad no puede interrumpir ni a una de alta prioridad ni a otra de baja prioridad.

Si se usaran más de dos interrupciones por prioridad, entonces se tendría que saltar a dos rutinas diferentes desde cada dirección de los vectores de interrupción. En cada rutina se tendría que verificar la bandera que está en nivel '1' para identificar la causa de la interrupción y luego saltar a la rutina de servicio de interrupción correspondiente.

6.4 Latencia de una interrupción

Como se puede observar, las interrupciones permiten implementar el concepto de multitarea en el procesamiento de los datos por parte del CPU. De esta manera, se ejecutan diversos procesos que solo serán atendidos cuando sea necesario. Sin embargo, la ejecución del programa siempre es secuencial y, por lo tanto, las rutinas de interrupción también se ejecutarán de modo similar.

Una interrupción no es un proceso inmediato. Cuando una bandera de interrupción se activa, el PC toma el valor 0x08 o 0x018 dependiendo de la prioridad de la interrupción. El salto genera un retardo de 2 ciclos de instrucción. Además, el procesador guarda en la pila rápida el valor del registro W, STATUS y BSR, lo cual genera un ciclo de instrucción adicional. Luego, en el vector de interrupción se deberá colocar el salto a la rutina de servicio de interrupción, lo que genera 2 ciclos más de interrupción, sumados al tiempo de la rutina de servicio de interrupción completa, que, dependiendo de su complejidad, puede tomar varios ciclos de instrucción. Finalmente, esta rutina deberá terminar con el borrado de la bandera de interrupción y con la instrucción retfie, lo cual agrega 3 ciclos de instrucción adicionales.

En resumen, la latencia de una interrupción corresponde a los retardos inherentes que no pueden ser obviados por el programador. El retardo mínimo está formado por 3 ciclos de instrucción en el acceso al vector, luego, 2 ciclos por el salto a la rutina de interrupción y al menos tres ciclos de instrucción para borrar la bandera y salir de la rutina de servicio. Esto suma una latencia de 8 ciclos de instrucción.

Si al retornar de una interrupción se utiliza la instrucción retfie FAST, el valor del registro W, STATUS y BSR será recuperado de la pila rápida. La opción FAST no agrega ciclos adicionales a la latencia.

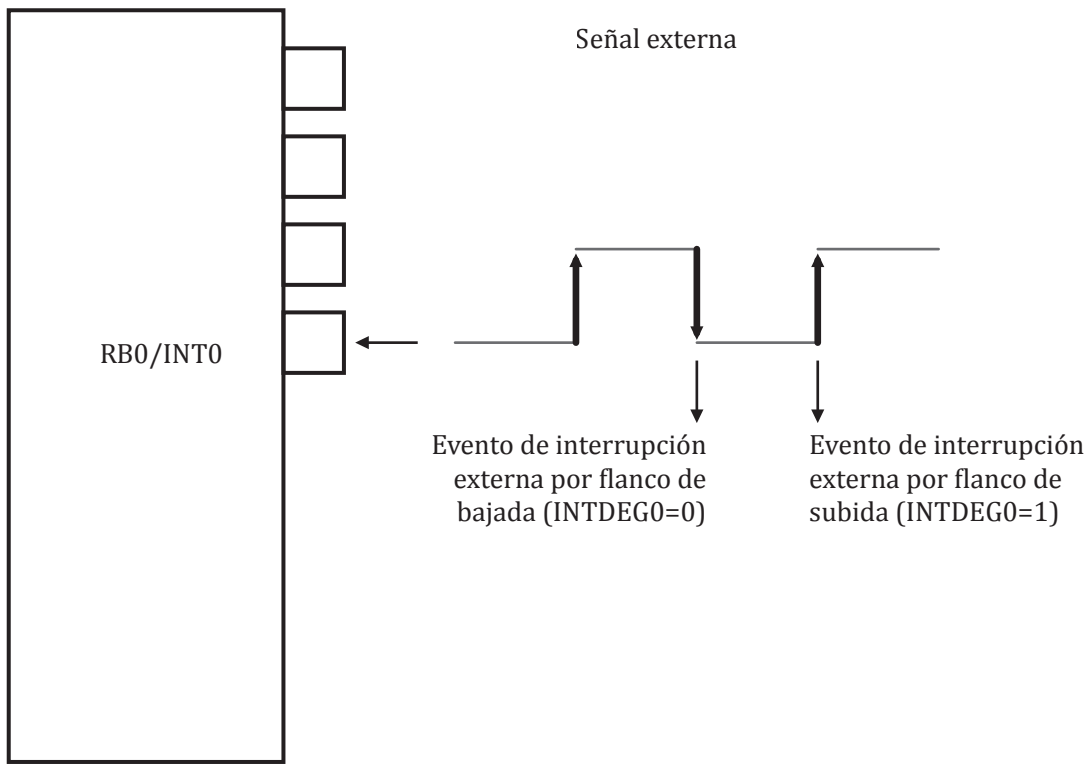
6.5 Ejemplo de configuración de una interrupción externa

El PIC18F4550 consta de tres pines de interrupción externa: INT0 (pin RB0), INT1 (pin RB1), INT2 (pin RB2). Cada uno de estos pines está preparado para recibir una señal cuadrada TTL y generar una interrupción cada vez que ocurra un flanco de subida o de bajada en la señal. Esto se observa en el esquema del gráfico 6.5.

La sensibilidad al flanco se selecciona con el bit INTDEG0 del registro INTCON2. Si este bit es '0', el módulo de interrupción externa será sensible al flanco de bajada, con lo cual el bit INT0IF del registro INTCON se pondrá a nivel '1' cuando ocurra este evento. Si, en cambio, el bit INTDEG0 se pone a '1', la sensibilidad se configura en el flanco de subida.

Un módulo de interrupción externa puede ser usado para múltiples aplicaciones. Por ejemplo, desde la detección de la pulsación de un pulsador, detección de señales de reloj en la comunicación serial síncrona, detección de velocidad de un motor mediante el análisis de la señal de un enconder y muchas otras aplicaciones más.

Gráfico 6.5. Conexión de una señal externa al pin RB0/INT0



Elaboración propia.

En el circuito esquemático del gráfico 6.6 se puede ver un pulsador conectado al pin RB0/INT0. Como se observa, el pulsador se conecta a tierra. La resistencia de *Pull Up* se habilita internamente en el Puerto B, por lo cual no es necesario colocar dicha resistencia externamente. Cada vez que se pulse el contacto, se generará un flanco de bajada de la señal detectada por RB0/INT0. Cuando el pulsador se suelte, se generará un flanco de subida.

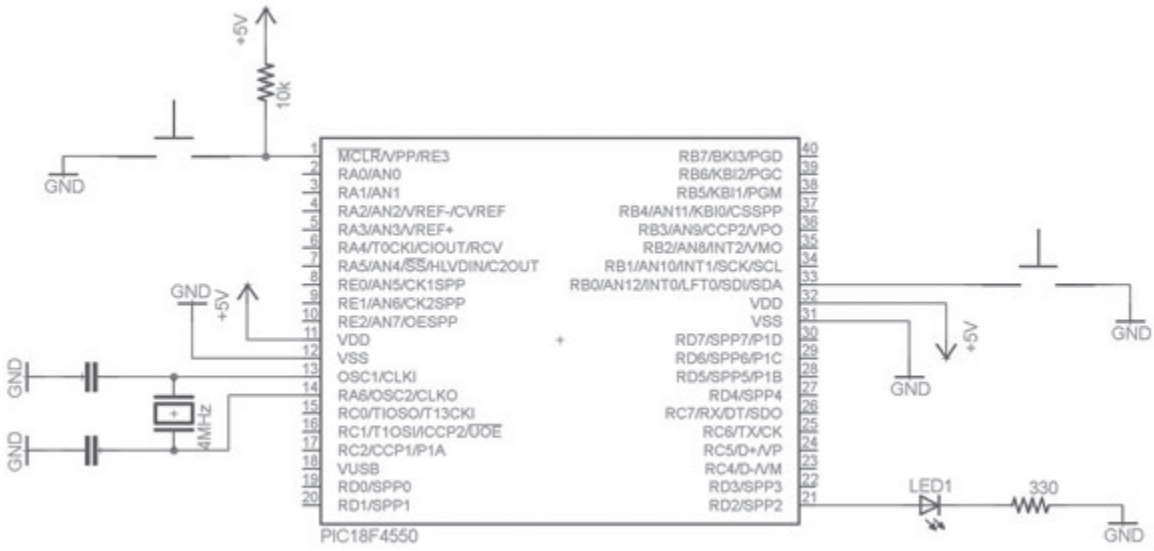
En el gráfico 6.6 se puede observar un LED conectado al pin RD2. Este LED oscilará entre prendido y apagado cada vez que se genere un flanco de bajada en el pin RB0/INT0. Dado que este evento depende de una pulsación, que es un proceso totalmente impredecible, se requiere del uso de una interrupción para su detección.

En el programa 6.4 se muestra el código completo. Se utiliza una única interrupción de alta prioridad, se configura la sensibilidad del pin RB0/INT0 en flanco de bajada y se activa la interrupción. Posteriormente, el programa se mantiene en un bucle indefinido. Cuando el evento externo ocurre, se ingresa al vector de interrupción de alta prioridad 0x08 y se ejecuta la rutina de servicio de interrupción, donde se invierte el estado del pin 2 del Puerto D y se borra la bandera de interrupción para luego retornar al bucle indefinido.

Programa 6.4. Ejemplo de detección de pulsaciones de un pulsador mediante la interrupción externa INT0

```
#include <P18F4550.INC>
        org 0x00
        goto MAIN
        org 0x008
        goto INT_INT0
        org 0x018
        retfie
        org 0x020
MAIN:
        setf TRISB              ;El Puerto B se configura como entrada digital.
        bcf INTCON2,RBPU        ;Se activan las resistencias de Pull Up internas del
                                ;Puerto B.
        bcf TRISD,2             ;Se configura el pin 2 del Puerto D como salida
                                ;digital.
        bcf INTCON2,INTEDG0     ;Se elige flanco de bajada para la INT0.
        bsf INTCON,INT0IE       ;Se activa la interrupción externa 0.
        bsf INTCON,GIE          ;Se habilita la máscara de interrupción.
INICIO:
        goto INICIO             ;Bucle indefinido.
INT_INT0:
        btg LATD,2              ;Se invierte el estado del pin 2 del Puerto D.
        bcf INTCON,INT0IF       ;Se pone a nivel '0' la bandera de interrupción.
        retfie                  ;Se retorna de la interrupción.
        END
```

Gráfico 6.6. Diagrama esquemático del ejemplo de configuración de la interrupción externa INT0



Elaboración propia.

Video

Cómo configurar la interrupción externa 0 (INT0) y de desbordamiento del Timer 0 con prioridades diferentes.

http://bit.ly/1EOvmB7

Resolución de videos en HD

Preguntas de repaso

1. Explique qué ocurre con el valor apuntado por el PC en un evento de interrupción.
2. ¿Qué entiende por prioridad de una interrupción?
3. ¿Cuántas prioridades maneja el PIC18F?
4. ¿Cómo se asignan las prioridades a una interrupción específica?
5. ¿Qué significado tiene la latencia de una interrupción?
6. Analice el siguiente código:

```
        org 0x00
        goto MAIN
        org 0x08
        goto RUT_PUERTOB
        org 0x018
        goto RUT_TMR0
        org 0x20
MAIN:
        bsf RCON,IPEN           ;PC= 0x20
        bcf INTCON2,TMR0IP      ;PC= 0x22
        bsf INTCON2,RBIP        ;PC= 0x24
        bsf INTCON,RBIE         ;PC= 0x26
        bsf INTCON,TMR0IE       ;PC= 0x28
        bsf INTCON,GIEL         ;PC= 0x2A
        bsf INTCON,GIEH         ;PC= 0x2C
INICIO:
        movlw '0'               ;PC=0x2E
        movwf 0x30,0            ;PC=0x30
        addlw 0x04              ;PC=0x32
        movwf 0x30,0            ;PC=0x34
        incf  0x30,f,0          ;PC=0x36
        goto INICIO             ;PC=0x38
RUT_TMR0:
        btfss INTCON,TMR0IF     ;PC=0x3C
        retfie                  ;PC=0x3E
        movlw .4                ;PC=0x40
        nop                     ;PC=0x42
        nop                     ;PC=0x44
        bcf   INTCON,TMR0IF     ;PC=0x46
        retfie                  ;PC=0x48
RUT_PUERTOB:
        btfss INTCON,RBIF       ;PC=0x4A
        retfie                  ;PC=0x4C
        movlw .5                ;PC=0x4E
        nop                     ;PC=0x50
        nop                     ;PC=0x52
        bcf  INTCON,RBIF        ;PC=0x54
        retfie                  ;PC=0x56
        END
```

Si el PC contiene el valor 0x34 y tanto las banderas TMR0IF como RBIF se ponen a nivel ‘1’ en simultáneo, ¿qué nuevo valor tomaría el PC?

Respuestas

Respuestas del capítulo 6:

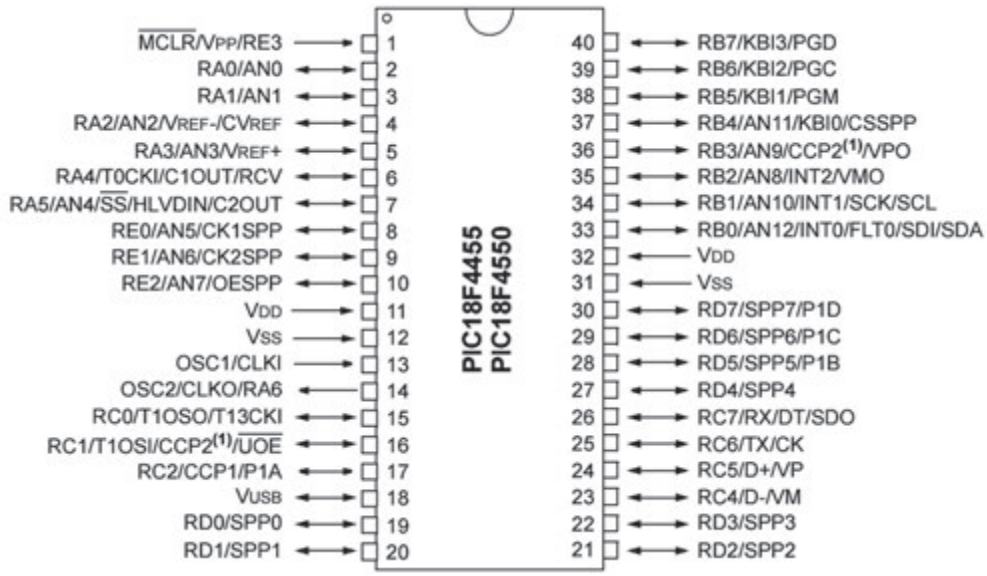
http://bit.ly/1Q4Y7Kt

Capítulo 7. Los puertos de entrada y salida

Los puertos de entrada y salida (E/S) del microcontrolador representan el medio que permite la interconexión de este dispositivo con el mundo externo. Gracias a los puertos E/S es posible controlar pantallas LCD alfanuméricas, teclados matriciales, lectores RFID, módems, sensores de temperatura, humedad, etc. El correcto conocimiento de las características de los puertos que ofrece el microcontrolador permite establecer qué tipo de periférico puede ser conectado a determinado puerto de E/S.

En el gráfico 7.1 se observa el diagrama de pines del PIC18F4550, que es el modelo de microcontrolador de la familia PIC18F usado para los ejemplos aplicativos de este libro. Aquí se pueden ver todos los pines del microcontrolador junto con las funciones adicionales que cumple cada pin.

Gráfico 7.1. Diagrama de pines del PIC18F4550



Fuente: http://www.microchip.com/downloads/en/devicedoc/39632e.pdf

7.1 Los puertos de entrada y salida del PIC18F4550

Como ya se explicó, todos los pines de entrada y salida del microcontrolador PIC18F4550 se encuentran agrupados en cinco puertos: Puerto A, Puerto B, Puerto C, Puerto D y Puerto E. En total se obtienen 24 pines de E/S. La configuración de cada pin está dada por tres registros de función especial:

1. TRISx (donde x puede ser A, B, C, D o E)
2. PORTx (donde x puede ser A, B, C, D o E)
3. LATx (donde x puede ser A, B, C, D o E)

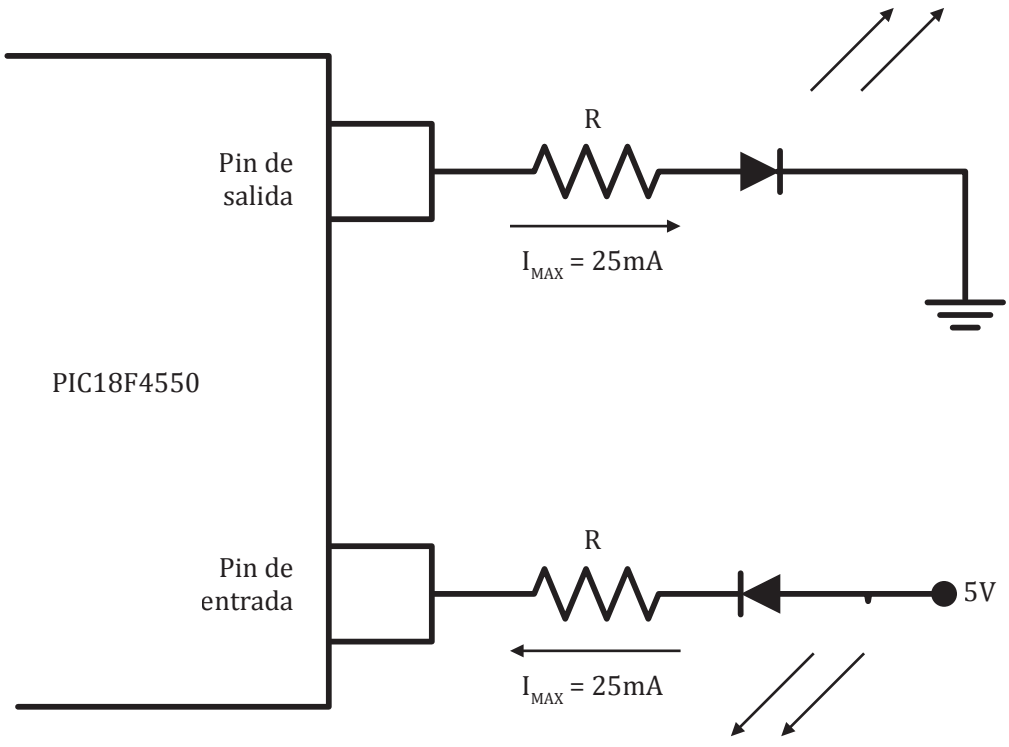
El funcionamiento de estos tres registros se explicó en los capítulos anteriores. Adicionalmente a estos, existen otros registros de función especial que operan directamente sobre ciertos pines cuando se han configurado los módulos periféricos internos.

Cada puerto se encuentra formado por una cantidad de pines distinta. Los puertos vienen asociados por su función especial. Por ejemplo, el Puerto A se especializa en servir de entrada para el conversor analógico digital. El Puerto B en cambio, presenta pines de interrupción externa que detectan flancos activos y cambios de estado de nivel, las líneas I^2C entre otros.

Los registros TRISx, PORTx, LATx son de 8 bits, esto implica que algunas de sus posiciones quedan en desuso para ciertos puertos que no completan los ocho pines (como los puertos A y E).

Cada pin del microcontrolador puede generar una corriente máxima de 25 miliamperios y al mismo tiempo puede soportar una corriente de la misma magnitud. Esto implica que cada pin conectado como salida tiene la capacidad de accionar ciertos elementos como el encendido de LED o Displays 7 segmentos, pequeños *Buzzers*, activar transistores, entre otros. En el esquema del gráfico 7.2 se observa la máxima corriente generada y absorbida por cada pin.

Gráfico 7.2. Máxima corriente de entrada y salida generada por cada pin E/S



Elaboración propia.

Algunos pines ofrecen características especiales para diversas funciones. A continuación se muestran las diversas opciones que ofrece cada uno de los puertos del microcontrolador PIC18F4550.

7.1.1 El Puerto A

El Puerto A se compone de 7 pines de E/S, de los cuales RA6 cumple también con la función de ser el pin de entrada para el oscilador externo (OSC2). La función principal de este puerto es la de servir como los canales de entrada analógica digital para el convertidor interno del microcontrolador, a excepción del pin RA4, que sirve como entrada de reloj externa para una señal que puede controlar el tiempo de conteo del temporizador TMR0.

Los pines RA0 y RA1 (pines 2 y 3 del PIC18F4550), aparte de operar como entradas y salidas digitales, cumplen con la tarea de representar los canales 0 y 1 de entrada al convertidor analógico digital. En el modo entrada analógica es muy importante que la configuración del registro TRISA sea de entrada digital, ya que durante la conversión los pines se comportan como entradas analógicas, pero, una vez finalizada la conversión, respetarán la configuración del registro TRISA, pudiendo ocasionar un cortocircuito si las tensiones generadas por la salida del pin y la señal analógica fueran diferentes (ya que el pin configurado como salida digital puede generar una tensión diferente a la señal analógica de entrada). Adicionalmente, existe un registro de nombre ADCON1 que permite configurar cuáles pines del Puerto A estarán configurados para entrada analógica y cuáles para modo digital. Este registro se mostrará en detalle en el capítulo 10, cuando se vea el funcionamiento del convertidor analógico digital.

Por otro lado, los pines RA2 y RA3 pueden operar en tres modos diferentes, además de ser configurados como entrada y salida digital. Estos pines corresponden a los canales 2 y 3 del convertidor analógico digital. Adicionalmente, el pin RA2 puede operar como la tensión de referencia negativa para el convertidor analógico digital o también como el umbral de tensión bajo para el modo comparador analógico. En cuanto al pin RA3, este puede operar como la tensión de referencia positiva para el convertidor analógico digital. Este modo de trabajo para estos pines se configura en el registro ADCON1.

El pin RA4 es un caso especial que no cumple con una función de entrada en el convertidor analógico digital, sino que más bien sirve como entrada de reloj para el módulo de temporización TIMER 0. Ingresando una señal cuadrada por este pin, se logra que el registro TMR0 se incremente en cada flanco de subida o de bajada dependiendo de la configuración del bit TOSE del registro T0CON. Además, puede actuar como salida del primer comparador en el modo de comparación. Adicionalmente, el pin RA4 cumple con la función especial de servir como entrada de señal para la conexión de un transceptor USB externo cuando se utiliza este dispositivo con el módulo USB del PIC18F4550.

El pin RA5 actúa como canal 4 del convertidor analógico digital y puede operar como el pin SS (*Slave Select* o selector de esclavo) del módulo MSSP o también como la salida del segundo comparador en el módulo de comparación. Queda sobreentendido que, además, este pin funciona como entrada y salida digital.

El pin RA6 es un caso especial. Normalmente, este pin funciona como entrada de reloj externo, tanto para el cristal como para una señal de reloj cuadrada que puede seleccionarse en los modos de reloj EC y ECPLL. Sin embargo, si se eligen modos de reloj con oscilador interno, este pin se puede aprovechar como entrada y salida digital extendiendo el tamaño del Puerto A en 7 bits.

Los registros de función especial asociados al Puerto A son los siguientes: LATA, PORTA, TRISA, ADCON1, CMCON, CVRCON, UCON.

7.1.2 El Puerto B

El Puerto B se compone de 8 pines de E/S. Todos estos pines poseen una resistencia de *Pull Up* interno que puede ser habilitado o deshabilitado en todos los pines de este puerto. Las resistencias de *Pull Up* se habilitan con el bit RBPU del registro INTCON2. Con este bit a nivel '0' lógico las resistencias quedan activas, mientras que con nivel '1' las resistencias se desactivan. Cuando uno de los pines del Puerto B se configura como salida digital, la resistencia de *Pull Up* queda desactivada en ese pin.

Durante el encendido del microcontrolador, los pines RB4-RB0 se comportan como entradas analógicas, mientras que los pines RB7-RB5 se configuran como entradas digitales por defecto. Es recomendable, cuando se configure el Puerto B como salida, que los registros PORTB y LATB sean inicializados en el valor 0x00 antes de configurar el registro TRISB.

Los pines RB7-RB4 poseen la opción de generar una interrupción cada vez que cambian de estado lógico. Evidentemente, estos pines deben estar configurados como entradas digitales para esto. Cada vez que el valor actual del PORTB es diferente del valor anterior, se produce un evento de interrupción disparado por el bit RBIF del registro INTCON. Esta condición se logra eliminar leyendo el registro PORTB o, en todo caso, borrando la bandera RBIF. Este modo de operación puede despertar al CPU del PIC18F4550 del estado de bajo consumo.

Adicionalmente, los pines RB7 y RB6 operan las líneas de datos (PGD) y reloj (PGC) para el modo de depuración y grabación del microcontrolador mediante el programador ICD2 (en los cuales se incluyen el PICkit 2 y PICkit 3). Por otro lado, la línea RB4 puede operar como el canal 11 de entrada para el convertidor analógico digital y, a su vez, opera como salida de control del módulo CSSPP.

El pin RB3 puede operar como entrada o salida digital y como canal 9 de entrada al convertidor analógico digital. Además, este pin opera como generador de PWM, salida de comparación o entrada de captura para el módulo CCP2.

Por otro lado, el pin RB2 actúa como entrada o salida digital y como canal 8 de entrada al convertidor analógico digital. Además, este pin sirve como entrada para la Interrupción externa 2 (INT2) y también para manejar la señal VMO de un transceptor USB externo.

El pin RB1 opera como entrada o salida digital y además representa el canal 10 de entrada al convertidor analógico digital. Adicionalmente, esta línea se puede configurar como entrada para la Interrupción externa 1 (INT1) y como señal de reloj para los modos de comunicación serial SPI (línea SCK) e I^2C (línea SCL).

Finalmente, el pin RB0 es el que tiene más funciones asociadas. Puede operar como entrada o salida digital y además representar el canal 12 de entrada al convertidor analógico digital. También puede operar como entrada para la Interrupción externa 0 (INT0) y como entrada para una señal de PWM, y sirve como entrada de datos para los modos de comunicación serial SPI (línea SDI) e I^2C (línea SDA).

Debido a la gran cantidad de funciones que cumplen los pines del Puerto B, se tienen varios registros de función especial asociados al funcionamiento de cada pin. Estos registros son los siguientes: LATB, PORTB, TRISB, INTCON, INTCON2, INTCON3, ADCON1, SPPCON, SPPCFG y UCON.

7.1.3 El Puerto C

El Puerto C se compone de 7 pines E/S que, además, cumplen con otras funciones especiales, dedicadas exclusivamente a las comunicaciones seriales que contiene el microcontrolador y también a los modos de entrada y salida de captura.

Los pines RC6 y RC7 operan como entradas y salidas digitales. Adicionalmente, pueden servir como pines de transmisión y recepción serial del módulo EUSART, respectivamente. En el caso del pin RC7, este también puede actuar como la línea de salida de datos (SDO) del módulo SPI.

Los pines RC5 y RC4 son utilizados para la comunicación USB y no pueden ser configurados como salida digital. Estos pines únicamente operan como entradas digitales. Su única función adicional es la de operar como las líneas de datos D- (RC4) y D+ (RC5) del bus USB. Es a través de estos pines que el microcontrolador puede transmitir y recibir datos por medio del protocolo USB.

El pin RC3 en el PIC18F4550 no se encuentra implementado. En su lugar se encuentra el pin V_{USB}, que es una salida de tensión regulada a 3.3 V y que debe ser conectada a un capacitor de desacoplo. El pin RC2 actúa como entrada y salida digital y, además, permite la entrada de captura y comparación de salida del módulo CCP1. Cuenta también con la opción de generar una señal de PWM mejorada.

El pin RC1 también cumple con la función de servir para la conexión del cristal de baja frecuencia. En esta configuración (T1OSI), el pin actúa como entrada para dicha señal de reloj. Además, sirve como pin generador de PWM o de entrada de captura para el módulo de entrada de captura y comparación de salida CCP2.

El pin RC0 actúa como entrada y salida digital. Además, puede operar como entrada para una señal de reloj externa encargada de controlar la cuenta del temporizador *Timer 3* y *Timer 1* (modo T13CKI). Por otro lado, este pin puede actuar como salida para la conexión de un cristal de baja frecuencia como el de 32,768 kHz, para que permita el ingreso de una señal para un reloj en tiempo real.

Finalmente, los registros asociados al funcionamiento de los pines del Puerto C son los siguientes: LATC, PORTC, TRISC y UCON.

7.1.4 El Puerto D

El Puerto D es el más homogéneo de todo el microcontrolador. Esto implica que sus pines cumplen con muy pocas funciones adicionales aparte de operar como entrada o salida digital. Este puerto, al igual que el Puerto B, contiene resistencias de *Pull Up* internas que pueden ser habilitadas configurando el bit RDPU del registro PORTE (bit 7) en nivel '0'.

El Puerto D tiene como función principal operar en el modo SPP (de las siglas en inglés *Streaming Parallel Port*), en el cual sus salidas TTL permiten enviar un valor de 8 bits a otros dispositivos con una alta tasa de transferencia. Los pines RD5, RD6 y RD7, adicionalmente, pueden actuar como generadores de PWM.

Los registros asociados al funcionamiento del Puerto D son los siguientes: LATD, TRISD, PORTD, PORTE, CCP1CON, SPPCON.

7.1.5 El Puerto E

El Puerto E puede trabajar en modo de 3 pines (por defecto) o en modo de cuatro pines. El pin RE3 también cumple con la función de ser el pin de entrada de pulso de Reset (MCLR) del microcontrolador. Cuando el bit MCLRE del registro de configuración se pone a nivel '0', el pin RE3 actúa únicamente como entrada digital, obviando el contenido del registro TRISE y LATE.

El pin RE0 puede operar como entrada o salida digital y, adicionalmente, puede configurarse como el canal 5 de entrada al convertidor analógico digital. Además, este pin funciona como la salida de reloj 1 para el módulo SPP.

Por otro lado, el pin RE1 opera de entrada y salida digital como el canal 6 de entrada al convertidor analógico digital. Además, actúa como la salida de reloj 2 para el módulo SPP.

El pin RE2 opera como entrada y salida digital al mismo tiempo que puede configurarse como el canal 7 de entrada del convertidor analógico digital. Además, opera como entrada de habilitación del módulo SPP.

Los registros asociados al funcionamiento del Puerto E son los siguientes: PORTE, LATE, TRISE, ADCON1, CMCON, SPPCON y SPPCFG.

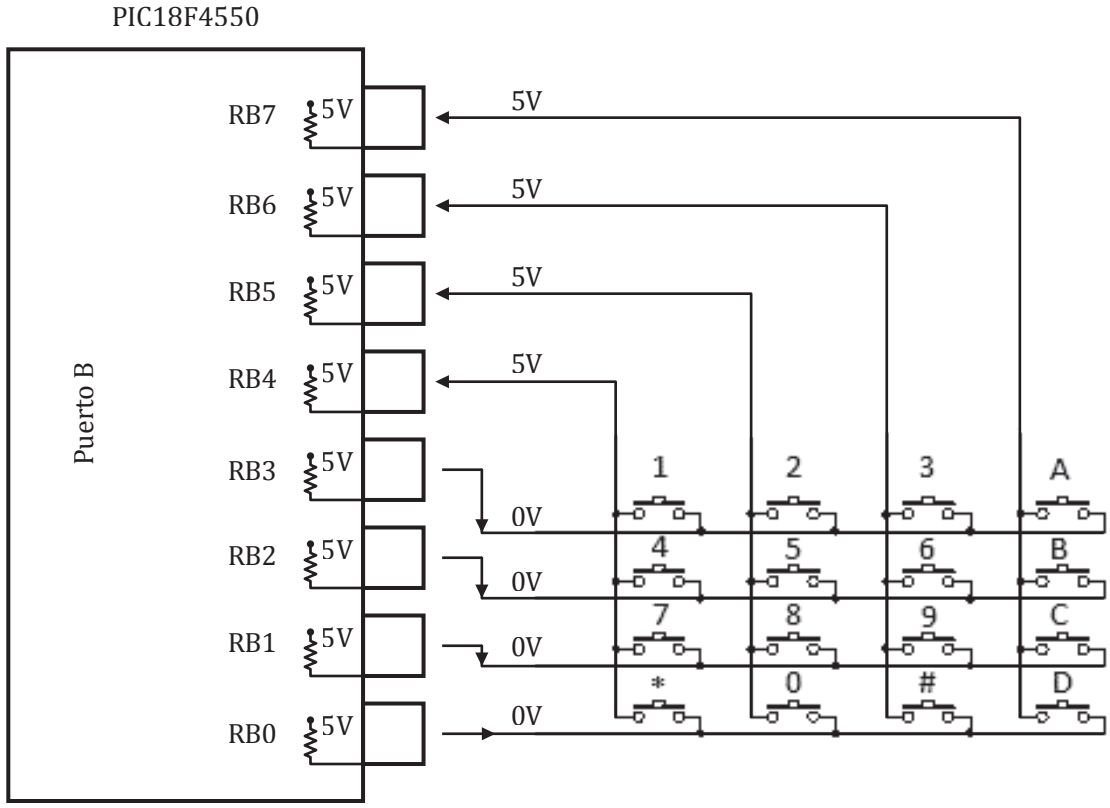
7.2 Ejemplo de control de un teclado matricial mediante la interrupción de cambio de estado del Puerto B

En el capítulo 5 se vio un ejemplo de detección de teclas pulsadas de un teclado matricial conectado al Puerto B. Este algoritmo, si bien funciona y puede ser probado en el microcontrolador, no resulta muy eficiente, ya que debe estar continuamente verificando el estado del Puerto B para detectar la pulsación de una tecla. Esto genera un desgaste de recursos y de energía innecesario para el procesador.

En el capítulo 6 se analizó el concepto de interrupción y se mencionó que el Puerto B posee la opción de generar una interrupción por el cambio de estado de los pines RB4-RB7. Esto representa una condición significativa para mejorar la rutina de detección de teclas pulsadas.

En el diagrama del gráfico 7.3 se puede encontrar la condición inicial para la detección de teclas pulsadas. Si los pines RB4-RB7 se configuran como entradas digitales con *Pull Up* activo, entonces sus pines estarán en nivel '1', mientras no haya una tecla pulsada. Luego de esto, se habilitará la interrupción por el cambio de estado del Puerto B, de manera que, si ocurriera un cambio en alguno de los pines RB4-RB7, se generaría una interrupción. Los pines RB0-RB3 conectados a las filas del teclado se configuran como salidas digitales a nivel '0', lo cual garantiza que, de pulsarse una tecla, uno de los pines RB4-RB7 cambiará a nivel '0'.

Gráfico 7.3. Condición inicial para la detección de teclas pulsadas en el teclado matricial



Elaboración propia.

Al detectarse un '0' en uno de los pines RB4-RB7 se generará una interrupción. En la rutina de servicio de interrupción se deberá determinar cuál de estos pines posee el nivel '0' realizando una lectura sobre el registro PORTB. Dependiendo del pin que contenga este nivel bajo, se determinará la columna involucrada en la pulsación de la tecla. Posteriormente, se cambiará la configuración del Puerto B, modificando los pines RB4-RB7 a salidas digitales con nivel '0' y los pines RB0-RB3 como entradas con *Pull Up* activo. Inmediatamente se vuelve a realizar la lectura del registro PORTB y se analiza cuál de los pines RB0-RB3 contiene el nivel '0'. Con esto se habrá detectado la intersección de la fila y columna de la tecla pulsada.

Posteriormente a la detección, es necesario retornar la configuración de los pines RB4-RB7 y RB0-RB3 a su condición inicial y esperar al menos 100 milisegundos para darle tiempo al usuario de soltar la tecla y volver a esperar la siguiente interrupción. Además, este tiempo es necesario para ignorar el efecto rebote de la pulsación de la tecla. Cabe mencionar que, si el usuario no suelta la tecla y la mantiene pulsada, la interrupción no finalizará hasta que la tecla sea soltada.

En el programa 7.1 se muestra el código que realiza la detección de teclas pulsadas del teclado matricial haciendo uso de la interrupción por cambio de estado de los pines RB4-RB7. El algoritmo inmediatamente asigna el código ASCII del valor de la tecla impreso en su respectiva posición (observar los dígitos asociados a las teclas en el gráfico 7.3). Esto permitirá más adelante mostrar ejemplos de interacción de esta rutina con el control de la pantalla LCD alfanumérica.

Programa 7.1. Algoritmo de detección de teclas pulsadas haciendo uso de la interrupción de cambio de estado de los pines RB4-RB7 del Puerto B

```
#include <P18F4550.INC>
        org 0x00
        goto MAIN
        org 0x08
        goto RUT_INT
        org 0x018
        retfie
        org 0x020
MAIN:
        movlw 0xF0
        movwf TRISB
        clrf INTCON2
        clrf  LATB
        clrf TRISD
        bsf  INTCON,RBIE
        bsf  INTCON,PEIE
        bsf  INTCON,GIE
INICIO:
      movlw .0
      xorwf tecla,W
      btfsc  STATUS,Z
      goto INICIO
      movff tecla,LATD
      clrf     tecla
      goto   INICIO

RUT_INT:
        btfss INTCON,RBIF
        retfie
        btfss PORTB,7
        goto  COLUMNA4
        btfss PORTB,6
        goto  COLUMNA3
        btfss PORTB,5
        goto  COLUMNA2
        btfss PORTB,4
        goto  COLUMNA1
        retfie
```

```
COLUMNA1:
        movlw 0x0F
        movwf TRISB
        clrf  LATB
        btfss PORTB,3
        movlw '1'
        btfss PORTB,2
        movlw '4'
        btfss PORTB,1
        movlw '7'
        btfss PORTB,0
        movlw '*'
        movwf tecla
        goto  SALIR
COLUMNA2:
        movlw 0x0F
        movwf TRISB
        clrf  LATB
        btfss PORTB,3
        movlw '2'
        btfss PORTB,2
        movlw '5'
        btfss PORTB,1
        movlw '8'
        btfss PORTB,0
        movlw '0'
        movwftecla
        goto  SALIR
COLUMNA3:
        movlw 0x0F
        movwf TRISB
        clrf  LATB
        btfss PORTB,3
        movlw '3'
        btfss PORTB,2
        movlw '6'
        btfss PORTB,1
        movlw '9'
        btfss PORTB,0
        movlw '#'
        movwf tecla
        goto  SALIR
```

```
COLUMNA4:
        movlw 0x0F
        movwf TRISB
        clrf  LATB
        btfss PORTB,3
        movlw 'A'
        btfss PORTB,2
        movlw 'B'
        btfss PORTB,1
        movlw 'C'
        btfss PORTB,0
        movlw 'D'
        movwftecla
        goto  SALIR
SALIR:
        call  DELAY100MSEG
        movlw 0xF0
        movwf TRISB
        clrf  LATB
        bcf   INTCON,RBIF
        retfie
```

7.3 Manejo de un módulo LCD alfanumérico basado en el controlador HD44780

Aunque los Displays de 7 segmentos son muy fáciles de usar, estos son grandes y limitados en la cantidad de formas de caracteres que pueden presentar. Cuando una aplicación requiere mostrar más que unos cuantos números y unas cuantas letras, los Displays de 7 segmentos se vuelven inadecuados. Los displays de cristal líquido (LCD) son una excelente alternativa cuando una aplicación requiere mostrar una gran variedad de caracteres alfanuméricos. Un display de cristal líquido posee las siguientes ventajas: alto contraste, bajo consumo de energía y ocupa poco espacio. Además, permite mostrar tanto caracteres alfanuméricos como gráficos simples.

En el gráfico 7.4 se muestra un módulo LCD alfanumérico típico. Este módulo LCD está compuesto de una pantalla de cristal líquido (LCD) y un controlador integrado conocido como el HD44780 (algunos diseños incluyen otro modelo de controlador, como el KS0066). En este tipo de display, la pantalla se divide en filas y columnas (con pequeños espacios que los separan). Algunos módulos poseen cuatro filas y otros incluyen hasta 20 y 40 columnas. Cada posición puede mostrar un carácter (alfanumérico o un gráfico predefinido por el usuario). Cada carácter se encuentra constituido por una matriz de puntos que puede ser de 5x8 o de 5x10 píxeles. Si bien es cierto, no es posible direccionar cada píxel, pero sí se puede crear un nuevo carácter asignándole los píxeles que estarán activos e inactivos dentro de la matriz.

Gráfico 7.4. Imagen de un LCD alfanumérico típico de 2 filas y 16 columnas

Elaboración propia.

Para interconectar el controlador HD44780 o el KS0066 a un microcontrolador es necesario conocer cómo se distribuye el diagrama de pines del bus de comunicación del módulo LCD alfanumérico. En el gráfico 7.5 se muestran los pines del módulo LCD alfanumérico. En total se tienen 11 líneas de comunicación hacia el microcontrolador, de las cuales tres corresponden al bus de control y 8 al bus de datos. Las líneas de datos son bidireccionales, permitiendo operaciones de escritura y lectura desde el sistema embebido. Además, se tiene una línea de alimentación (5 V), una de tierra y otra para conectar un voltaje entre 0.2 y 0.7 V que permite regular el nivel de contraste. Algunos módulos LCD presentan un diodo LED conectado en la parte posterior que sirve como luz de fondo. Esto adiciona dos pines más al bus de comunicaciones de los cuales uno debe ir a tierra y el otro a 5 V.

Gráfico 7.5. Diagrama de pines del módulo LCD alfanumérico

Número de pin	Símbolo	Entrada/Salida	Función
1	VSS		Tierra (GND)
2	VCC		Alimentación (5V)
3	VEE		Ajuste de contraste
4	RS	Entrada	Selector de modo de operación
5	R/W	Entrada	Modo lectura/ escritura
6	E	Entrada	Pin de habilitación
7	DB0	Entrada/Salida	Línea de datos LSB
8	DB1	Entrada/Salida	Línea de datos 1
9	DB2	Entrada/Salida	Línea de datos 2
10	DB3	Entrada/Salida	Línea de datos 3
11	DB4	Entrada/Salida	Línea de datos 4
12	DB5	Entrada/Salida	Línea de datos 5
13	DB6	Entrada/Salida	Línea de datos 6
14	DB7	Entrada/Salida	Línea de datos MSB

Elaboración propia.

Los pines DB7-DB0 son usados para intercambiar datos con el microcontrolador. El pin E (*Enable*) permite el ingreso de la señal de activación del módulo. La señal R/W determina la dirección de la transferencia de datos (lectura con nivel '1' lógico o escritura con nivel '0'). La señal RS selecciona el modo de operación a acceder. Cuando la señal en el pin RS se encuentra en nivel alto, se realizará una operación de acceso a memoria, mientras que con un nivel bajo se enviarán comandos de configuración. El pin V_{EE} es usado para controlar el brillo del Display y es frecuentemente conectado a un potenciómetro. El voltaje de entrada al pin V_{EE} no debería estar en su máximo valor (VCC) por un periodo de tiempo extendido para evitar dañar la pantalla LCD.

Un módulo LCD se puede interconectar directamente con un puerto E/S de un microcontrolador. En esta configuración el diseñador necesitará utilizar los pines de entrada y salida para controlar las señales E, R/W, RS y el bus de datos. El control del módulo LCD depende directamente de la manipulación de estas señales.

El HD44780 provee un conjunto de instrucciones para que el usuario pueda configurar los parámetros del LCD. Las operaciones realizadas por estas instrucciones se resumen en el gráfico 7.6. El significado de ciertos bits en estas instrucciones se explica en el gráfico 7.7. El HD44780 puede ser configurado para controlar la primera línea, la segunda línea, la tercera línea y la cuarta línea, en caso de que la pantalla posea una dimensión de cuatro filas.

Gráfico 7.6. Comandos de configuración del módulo LCD alfanumérico

Instrucción	Código										Tiempo de ejecución
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
Borrar Display	0	0	0	0	0	0	0	0	0	1	1.64 mseg
Cursor a Home	0	0	0	0	0	0	0	0	1	X	1.64 mseg
Configurar modo	0	0	0	0	0	0	0	1	I/D	S	40 µseg
Display encendido/ apagado	0	0	0	0	0	0	1	C	D	B	40 µseg
Desplazamiento del cursor	0	0	0	0	0	1	S/C	R/L	X	X	40 µseg
Conjunto de funciones	0	0	0	0	1	DL	N	F	X	X	40 µseg
Dirección CGRAM	0	0	0	1	Dirección de la CGRAM						40 µseg
Dirección DDRAM	0	0	1	Dirección de la DDRAM							40 µseg
Lectura de la bandera de ocupado	0	1	X	X	X	X	X	X	X	X	40 µseg
Escritura en CGRAM o DDRAM	1	0	Dirección de escritura de la DDRAM o CGRAM								40 µseg
Lectura de la CGRAM o DDRAM	1	1	Dirección de lectura de la DDRAM o CGRAM								40 µseg

Elaboración propia.

Un comando de configuración permite editar las opciones de funcionamiento del módulo LCD. Por ejemplo, desactivar el cursor o habilitar el parpadeo de este son configuraciones típicas. En el gráfico 7.7 se muestra la lista de configuraciones más utilizadas para el controlador HD44780.

Gráfico 7.7. Función de los bits de configuración del gráfico 7.6

Nombre del bit	Configuración	
I/D	'0' = Decrementar la posición del cursor.	'1' = Incrementar la posición del cursor.
S	'0' = El Display no se desplaza.	'1' = El Display se desplaza.
D	'0' = Display apagado	'1' = Display encendido
C	'0' = Cursor desactivado	'1' = Cursor activo
B	'0' = Parpadeo del cursor desactivado	'1' = Parpadeo del cursor activo
S/C	'0' = Desplazamiento del cursor	'1' = Desplazamiento del Display
R/L	'0' = Incremento a la izquierda	'1' = Incremento a la derecha
DL	'0' = Bus de datos de 4 bits	'1' = Bus de datos de 8 bits
N	'0' = 1 Fila activa	'1' = 2 filas activas
F	'0' = Matriz de 5x8 píxeles	'1' = Matriz de 5x10 píxeles
BF	'0' = Módulo LCD listo para recibir otro comando o dirección	'1' = Módulo LCD ocupado

Elaboración propia.

Cuando se accede a los registros de memoria, entonces es posible leer o escribir el módulo LCD. Este cuenta tres tipos de memoria, la DDRAM, CGRAM y CGROM. Las dos primeras son de lectura y escritura. En la DDRAM es posible fijar los valores que serán mostrados en la pantalla LCD, mientras que la CGRAM permite crear nuevos caracteres predefinidos por el usuario. La memoria CGROM, en cambio, contiene almacenados los formatos de todos los caracteres ASCII tradicionales y ASCII extendidos.

En modo lectura, el pin DB7 actúa como bandera de estado de ocupado. Mientras este pin se encuentra en estado '1', el controlador se encontrará procesando el último comando enviado por el sistema embebido. Cuando este pin se encuentre en nivel '0', el microcontrolador sabrá que ya es posible enviar un nuevo comando o dirección al controlador. Es importante que el pin DB7 se conecte a una línea que pueda actuar como entrada y salida del microcontrolador.

Es fundamental recalcar que el controlador HD44780 o el KS0066 pueden operar en dos modos de bus de datos: 8 bits o 4 bits. El último modo se utiliza cuando no se tienen muchos pines E/S del sistema embebido disponibles. La contraparte de este modo es que se requieren dos accesos para lograr acceder a la memoria o escribir un comando. En esta configuración, los pines DB4, DB5, DB6 y DB7 se convierten en el bus de datos, mientras que los pines DB0, DB1, DB2 y DB3 quedan inservibles.

El fabricante del controlador recomienda hacer uso de una secuencia específica de configuración al momento de encender la energía del módulo LCD:

1. Esperar 15 milisegundos después de que el voltaje de alimentación V_{DD} haya sido conectado y haber superado el 90% del nivel máximo de V_{DD}.
2. Enviar el dato 0x30 (DB4 = '1' y DB5 = '1') mientras RS = '0' y RW = '0' (modo escritura).
3. Esperar al menos por 5 milisegundos.
4. Volver a repetir el paso 2.
5. Esta vez esperar al menos 100 µsegundos.
6. Repetir nuevamente el paso 2.
7. Nuevamente se esperan al menos 100 µsegundos.
8. Repetir nuevamente el paso 2.

Una vez realizados estos ocho pasos, recién será posible iniciar la configuración del controlador. Aquí se deberán acceder a los registros del conjunto de funciones (donde se elegirá el modo de 4 bits de datos, 2 filas y matriz de 5x8 puntos), encendido del display, borrado de la pantalla y modo de entrada (desplazamiento del display y movimiento del cursor). Luego, solo quedará escribir los caracteres en la memoria DDRAM o crear nuevos caracteres en la CGRAM para mostrarlos en la pantalla LCD.

7.3.1 La memoria DDRAM

La memoria DDRAM (*Display Data RAM*) almacena los caracteres que van a ser vistos en la pantalla LCD. Cada dirección de la DDRAM hace referencia a una posición específica de la pantalla definida por una fila y columna.

El gráfico 7.8 muestra un mapa de direcciones para un display de 4 filas por 20 columnas. La primera columna comprende las direcciones de la 0 a la 19. La segunda columna empieza en las dirección 64 y culmina en la 83; mientras que la tercera columna continúa en la dirección 20 y culmina en la 39. Finalmente, la cuarta fila ocupa las direcciones 84 a la 103.

Gráfico 7.8. Direcciones de la memoria DDRAM

Línea	Direcciones																			
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
3	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
4	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103

Elaboración propia.

Normalmente, el cursor, de estar habilitado, estará ubicado en la posición indicada por la DDRAM. Usualmente la DDRAM se autoincrementa, lo que significa que una vez que se fija su dirección, la siguiente operación de escritura de un carácter se realizará en la siguiente dirección, con lo cual el nuevo carácter aparecerá a la derecha del anterior.

Cabe resaltar que este mapa de memoria es independiente del tamaño real del módulo LCD. Los LCD estándar poseen 2 filas por 16 columnas y presentan el mismo controlador que una pantalla de 4

filas por 20 columnas. El sistema embebido debe llevar la cuenta de las direcciones a las cuales accede y tiene que evitar tocar direcciones que corresponden a columnas que no se encuentran implementadas.

7.3.2 La memoria CGROM

La memoria de caracteres CGROM es de tipo no volátil y contiene 192 símbolos de 5 x 7 pixeles que son seleccionados con una palabra de 8 bits. Una vez definida la dirección de la DDRAM se escribe el código ASCII del carácter en el bus de datos, y con esto quedará impreso en dicha posición definida previamente.

Esta memoria facilita enormemente la programación de los mensajes en el microcontrolador, ya que la gran mayoría de lenguajes de programación de sistemas embebidos manejan la simbología de códigos ASCII.

7.3.3 La memoria generadora de caracteres CGRAM

En algunas ocasiones es necesario que el usuario genere uno o varios caracteres a su criterio. Por ejemplo, en algunas aplicaciones sería importante mostrar el logo de la empresa en la pantalla LCD. Dado que la CGROM solo contiene los caracteres estándar, entonces es necesario construir la forma de los caracteres. Para esto, el módulo LCD alfanumérico cuenta con una memoria CGRAM (*Character Generator RAM*) de 64 bytes que permite crear hasta 8 nuevos caracteres de 5x8 píxeles.

Cada carácter ocupa 8 bytes, lo que significa que está compuesto por 8 filas y 5 columnas. Tres bits de cada byte (los bits más significativos) no participan en la configuración del carácter. En el gráfico 7.9 se muestran los rangos de direcciones para definir cada uno de los 8 caracteres.

Gráfico 7.9. Rangos de direcciones de la memoria CGRAM para la creación de caracteres

Carácter	Dirección de inicio de la CGRAM	Dirección de fin de la CGRAM
1	0	7
2	8	15
3	16	23
4	24	31
5	32	39
6	40	47
7	48	66
8	56	63

Elaboración propia.

Se desea crear el carácter Δ para un tipo de mensaje a presentar en la pantalla. Dado que dicho carácter no existe en la lista de la CGROM, es necesario constituirlo en la CGRAM. Para esto, se debe descomponer Δ en una matriz de 8 filas por 5 columnas. Cada fila es un byte cuyos 3 bits más significativos no forman parte de la definición del carácter, tal como se muestra en el gráfico 7.10.

Gráfico 7.10. Diseño del carácter Δ

Dato	Bits							
	7	6	5	4	3	2	1	0
0x04	0	0	0	0	0	1	0	0
0x0A	0	0	0	0	1	0	1	0
0x11	0	0	0	1	0	0	0	1
0x11	0	0	0	1	0	0	0	1
0x11	0	0	0	1	0	0	0	1
0x11	0	0	0	1	0	0	0	1
0x11	0	0	0	1	0	0	0	1
0x1F	0	0	0	1	1	1	1	1

Elaboración propia.

Una vez creado el carácter como matriz, es necesario cargarlo en la memoria CGRAM. Lo que se debe hacer es crear una tabla de datos en el programa con los valores que se presentan en la columna Dato del gráfico 7.10. Luego, se debe acceder a una de las direcciones de inicio de un carácter en la CGRAM (direcciones 0, 8, 16, 24, 32, 40, 48 o 56). Si se quiere que Δ sea el carácter 4, entonces entre las direcciones 24 y 31 de la CGRAM se almacenarán los valores 0x04, 0x0A, 0x11, 0x11, 0x11, 0x11, 0x11 y 0x1F. Una vez hecho esto, para imprimir dicho carácter en la pantalla se deberá escribir el dígito 0x04 en el bus de direcciones fijado en una zona de la DDRAM.

7.3.4 Registros internos y tipos de instrucciones

Las instrucciones se seleccionan colocando el pin RS (*Register Select*) del módulo LCD a nivel '0'. Estas instrucciones permiten configurar el modo de operación de la pantalla LCD y pueden ser reconfiguradas en cualquier momento. El conjunto de instrucciones es el siguiente:

1. CLEAR DISPLAY. Esta función escribe el código 0x20 (Código ASCII del espacio en blanco) en todas las ubicaciones de la DDRAM. Luego, configura con el valor de 0 el contador de direcciones y retorna al display a su estado original. En otras palabras, la pantalla del display no muestra ningún carácter y el cursor se ubica en la esquina superior izquierda. También configura el bit I/D a nivel '1' (modo incremento).
2. RETURN HOME. Esta instrucción configura la dirección de la DDRAM en 0 dentro del contador de direcciones. El contenido de la DDRAM no cambia y el cursor se ubica en la esquina izquierda del display (primera fila y primera columna).
3. ENTRY MODE SET. El bit I/D de esta instrucción controla el incremento (I/D = 1) o decremento (I/D = 0) de la dirección DDRAM. El cursor se moverá a la izquierda o a la derecha dependiendo si este bit es '1' o '0'. Lo mismo se aplica a la escritura y lectura de la CGRAM.El bit S de esta instrucción controla el desplazamiento del display. El display se desplaza si S es '1'. En el otro caso, el display no se desplaza. Cuando S = '1' se daría la apariencia de que el cursor no se mueve y es

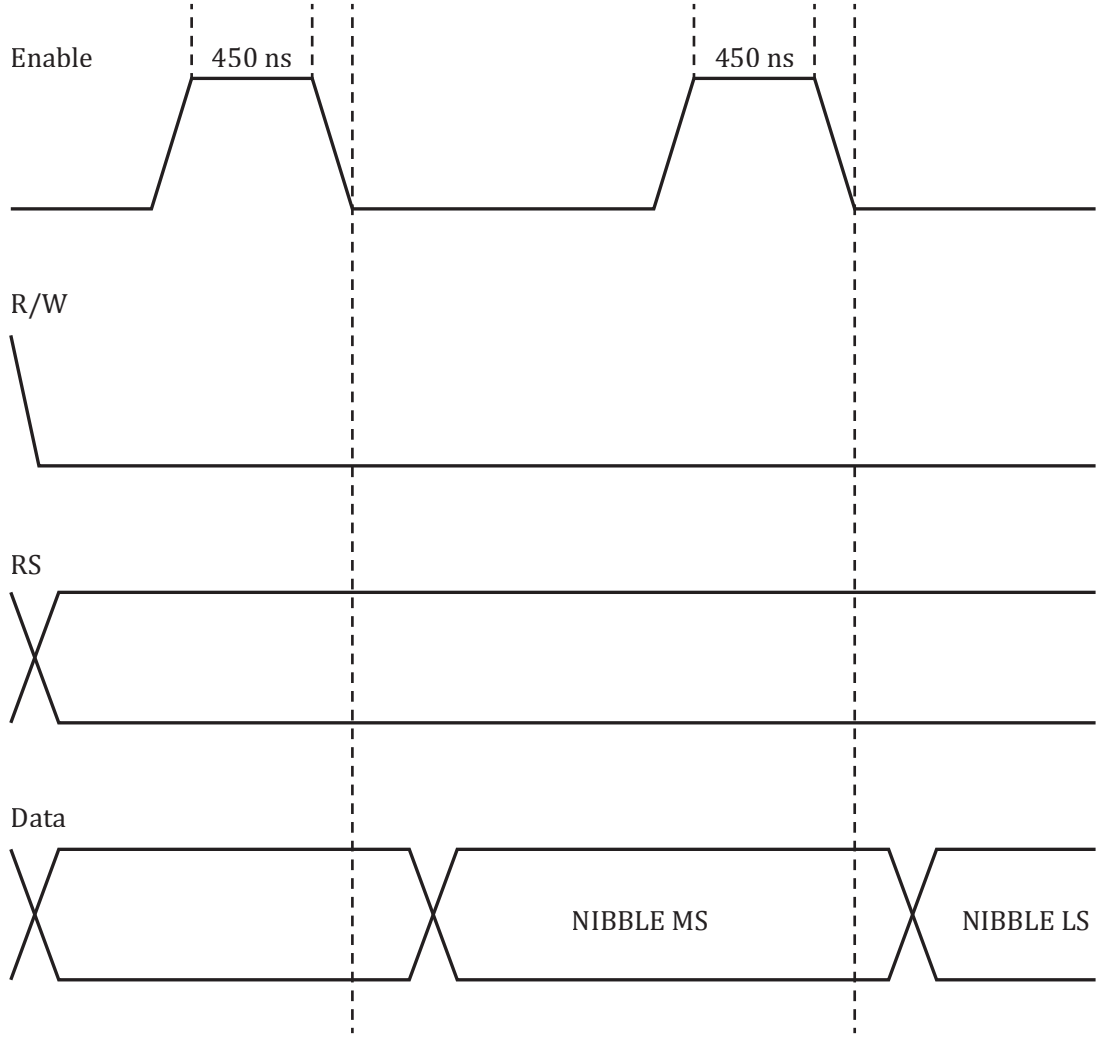
la pantalla del display la que se desplaza completamente. El display no se desplazará al hacer una lectura de la DDRAM o de la CGRAM.

4. DISPLAY ON/OFF CONTROL. Esta instrucción posee tres parámetros binarios: D, C y B. Cuando el bit D se encuentra a nivel '1', el display se enciende. El cursor se activa cuando el bit C es '1'. El carácter indicado por el cursor empezará a oscilar cuando el bit B (*Blink*) sea '1'.
5. DESPLAZAMIENTO DEL CURSOR. Esta instrucción desplaza la posición de cursor a la derecha o izquierda sin necesidad de escribir o leer los datos del display. El movimiento es controlado por dos bits como se muestra en el gráfico 7.6. Normalmente, esta función se utiliza para corregir un carácter o buscar una posición específica dentro del display. Cuando los datos mostrados son desplazados repetidamente, cada línea se mueve solo horizontalmente. La segunda línea del display no desplaza a la primera fila. El contenido del contador de direcciones no cambiará si la única acción desarrollada es desplazar el display.
6. CONJUNTO DE FUNCIONES. Esta instrucción permite al usuario configurar la longitud de los datos, seleccionar el número de líneas del display y seleccionar el tipo de letra. Existen tres bits de configuración en esta instrucción:DL: los datos se envían y reciben en una longitud de 8 bits (DB7 a DB0) cuando DL es '1' y una longitud de 4 bits (DB7 a DB4) cuando DL es '0'. N: este bit configura el número de líneas de datos. Cuando se configura en '0', el display muestra una sola línea.F: cuando se configura a '0', los caracteres trabajan en un formato de 5 x 8 puntos. Cuando se configura a '1', se selecciona el formato 5 x 10.
7. DIRECCIÓN DE LA CGRAM. Esta instrucción contiene la dirección de la CGRAM para ser configurada en el contador de direcciones.
8. DIRECCIÓN DE LA DDRAM. Esta instrucción permite al usuario configurar la dirección de la DDRAM en el contador de direcciones.
9. LECTURA DE LA BANDERA DE OCUPADO. Esta instrucción permite leer la bandera de ocupado (BF) y el contador de direcciones. La bandera BF indica cuando el controlador LCD se encuentra aún ejecutando una instrucción previa.

En el gráfico 7.11 se muestra el diagrama de tiempos para la escritura de un dato o comando en el módulo LCD. Como se observa, la línea E (*Enable*) es la que ejerce el primer control, teniéndose que sincronizar el envío de datos con esta línea. El pin R/W se mantiene continuamente a nivel '0', ya que se realiza un proceso de escritura y el pin RS va a determinar si se va a enviar un comando o una dirección.

Dado que el bus de datos es de 4 bits, la transferencia de los 8 bits de información del microcontrolador al módulo LCD se deberá dar en dos paquetes consecutivos. El primer envío coloca el *nibble* más significativo en el puerto de comunicaciones (pines DB4, DB5, DB6 y DB7). Una vez que el microcontrolador ha enviado un pulso de *Enable* (con una duración mínima de 450 nanosegundos), el controlador del módulo LCD procesará el primer *nibble*. Posteriormente, el microcontrolador colocará el *nibble* menos significativo y enviará el pulso de *Enable*, con lo cual el módulo LCD terminará de procesar el dato completo.

Gráfico 7.11. Diagrama de tiempos para la escritura de un comando o dirección en el módulo LCD haciendo uso de un bus de datos de 4 bits



Elaboración propia.

7.3.5 Control del HD44780 a través del PIC18F4550

Para poder hacer uso del módulo LCD de una manera ordenada y que permita entender la secuencia del programa, se propone el uso de una librería de rutinas que posibilite el manejo de una serie de funciones de la pantalla LCD, simplemente invocando a las etiquetas cuyo nombre especifica la función que cumple la rutina involucrada en dicha dirección de programa. En el programa 7.2 se muestra la librería LCD_LIB.asm que será agregada en los siguientes ejemplos presentados en el presente subcapítulo.

Programa 7.2. Librería para el manejo de las funciones del módulo LCD: LCD_LIB.asm

```
RS    EQU  0
RW   EQU  1
E     EQU  2
CBLOCK 0x000
          var
          dato
          rx
          aux1
          aux2
          aux3
          conta1
          conta2
          dir_CGRAM
          dir_CARACTER
ENDC
ORG 0x500

POS_CUR_FIL2:
          addlw 0xC0;
          movwf dato,0;
          call ENVIA_LCD_CMD
          return
POS_CUR_FIL1:
          addlw 0x80;
          movwf dato,0;
          call ENVIA_LCD_CMD
          return
DISPLAY_ON:
          movlw b'00001111'
          movwf dato,0;
          call ENVIA_LCD_CMD
          return
DISPLAY_OFF:
          movlw b'00001000'
          movwf dato,0;
          call ENVIA_LCD_CMD
          return
```

```
CURSOR_HOME:
        movlw b'00000010'
        movwf dato,0;
        call ENVIA_LCD_CMD
        return
CURSOR_OFF:
        movlw b'00001100'
        movwf dato,0;
        call ENVIA_LCD_CMD
        return
CURSOR_ON:
        movlw b'00001111'
        movwf dato,0;
        call ENVIA_LCD_CMD
        return
ENVIA_CHAR:
        movwf dato,0
        call ENVIA_LCD_DATO
        return
BORRAR_LCD:
        movlw 0x01
        movwf dato,0
        call ENVIA_LCD_CMD
        return
LCD_CONFIG:
        movlw 0x30
        movwf dato,0
        bcf   LATD,RS
        bcf   LATD,RW
        call  ENVIA_NIBBLE
        call  DELAY2MSEG
        call  DELAY2MSEG
        call  DELAY2MSEG
        movlw 0x30
        movwf dato,0
        call ENVIA_NIBBLE
call DELAY100uS
        movlw 0x30
        movwf dato,0
        call ENVIA_NIBBLE
        call DELAY100uS
```

```
        movlw 0x30
        movwf dato,0
        call ENVIA_NIBBLE
        movlw 0x20
        movwf dato,0
        call ENVIA_NIBBLE
        movlw 0x28          ;Conjunto de funciones (4 bits,2 lineas 5x8).
        movwf dato,0
        call ENVIA_LCD_CMD;
        movlw 0x08          ;Control del Display (Activar,mostrar cursor y que
                            ;parpadee).
        movwf dato,0
        call ENVIA_LCD_CMD
        movlw 0x01          ;Conjunto de inicio (Incrementar posición sin
                            ;desplazamiento)
        movwf dato,0
        call ENVIA_LCD_CMD
        movlw 0x06          ;Limpiar el Display;
        movwf dato,0
        call ENVIA_LCD_CMD
        movlw 0x0E          ;Limpiar el Display;
        movwf dato,0
        call ENVIA_LCD_CMD
        movlw 0x0C          ;Limpiar el Display;
        movwf dato,0
        call ENVIA_LCD_CMD
        return
ENVIA_NIBBLE:
        movwf rx,0
        movlw 0x0F
        andwf LATD,f,0
        movf  rx,W,0
        iorwf LATD,f,0
        bsf   LATD,E,0
        bcf   LATD,E,0
        return
ENVIA_LCD_CMD:
        bcf LATD,RS,0
        call LEER_LCD
        movlw b'00001000'
        movwf TRISD,0
```

```
        call DELAY100uS
        bcf  LATD,RW
        bcf  LATD,E
        bcf  LATD,RS
        movf dato,0,0
        andlw 0xF0;
        call ENVIA_NIBBLE
        swapf dato,f,0
        movf dato,0,0
        andlw 0xF0
        call ENVIA_NIBBLE
        return
LEER_LCD:
        movlw 0xF8
        movwf TRISD
        bcf   LATD,RS
        bsf   LATD,RW
        bsf   LATD,E
        bcf   LATD,E
        movf  PORTD,W,0
        movwf rx,0
        bsf   LATD,E
        bcf   LATD,E
        btfss rx,7,0
        return
        goto LEER_LCD
ENVIA_LCD_DATO:
        bsf  LATD,RS
        call LEER_LCD
        movlw b'00001000'
        movwf TRISD,0
        call DELAY100uS
        bcf  LATD,RW
        bcf  LATD,E
        bsf  LATD,RS
        movf dato,0,0
        andlw 0xF0;
        call ENVIA_NIBBLE
        swapf dato,f,0
        movf  dato,0,0
        andlw 0xF0
        call ENVIA_NIBBLE
```

```
        return
GENERA_CHAR_CGRAM:
        movlw 0x40
        addwf dir_CGRAM,W
        movwf dato
        call ENVIA_LCD_CMD
REPITE:
        TBLRD*+;
        movf  TABLAT,W
        movw fvar
        movlw 0xFF
        subwf var,W
        btfsc STATUS,Z
        goto  SALIR
        movff var,dato
        call ENVIA_LCD_DATO
        goto  REPITE
SALIR:
        movlw 0x80
        movwf dato
        call ENVIA_LCD_CMD
        return
```

En resumen, las funciones de cada subrutina son las siguientes:

1. POS_CUR_FIL2: esta rutina toma el valor cargado en el registro W y ubica el cursor en la segunda fila y en la columna especificada por el valor de W.
2. POS_CUR_FIL1: esta rutina toma el valor cargado en el registro W y ubica el cursor en la primera fila y en la columna especificada por el valor de W.
3. DISPLAY_ON: esta rutina activa la pantalla LCD, de manera que se puedan visualizar los caracteres alfanuméricos.
4. DISPLAY_OFF: esta rutina desactiva la pantalla LCD, impidiendo la visualización de caracteres alfanuméricos; de esta manera, el consumo de energía será mínimo.
5. CURSOR_HOME: después de ejecutar esta rutina, el cursor se ubicará en la primera fila y primera columna. Los caracteres mostrados en la pantalla no sufrirán mayor alteración.
6. CURSOR_OFF: el cursor es muy útil cuando se va a ingresar datos por un teclado y el dígito será visualizado en la pantalla. El cursor permite visualizar la posición donde el carácter aparecerá al pulsar la tecla. En algunas situaciones, cuando solo se van a desplegar mensajes, no es necesario visualizar el cursor.
7. CURSOR_ON: esta rutina habilita la presencia del cursor en la pantalla LCD.
8. ENVIA_CHAR: es una rutina muy importante, ya que permite imprimir los caracteres en la pantalla sobre la posición donde se ubica el cursor. Inmediatamente, el cursor se incrementa en una posi-

ción. El valor del carácter a escribir en la pantalla se debe cargar en el registro W previamente al llamado de la rutina.

9. BORRAR_LCD: esta rutina elimina el contenido mostrado en la pantalla y ubica el cursor en la primera fila y primera columna.
10. LCD_CONFIG: esta rutina inicializa el módulo LCD para que opere con un bus de datos de 4 bits, active el cursor con desplazamiento a la derecha, pantalla LCD activa y uso de 2 filas.
11. ENVIA_NIBBLE: es una rutina que se encarga de separar el byte de datos a enviar en su *nibble* más significativo y en el menos significativo. Luego, transmite cada *nibble* al módulo LCD, respetando el diagrama de tiempos del gráfico 7.11. Esta rutina es interna; es decir, es invocada automáticamente por las rutinas ENVIA_LCD_CMD y ENVIA_LCD_DATO.
12. ENVIA_LCD_CMD: esta rutina envía una instrucción al módulo LCD, cuyo código binario se debe cargar previamente en el registro de trabajo W.
13. LEER_LCD: es una rutina interna, cuya función consiste en esperar que el bit DB7 se ponga a nivel '0' lógico. indicando que el módulo LCD se encuentra listo para recibir una nueva dirección o comando. Cabe resaltar que es una rutina interna usada por los subprogramas ENVIA_LCD_CMD y ENVIA_LCD_DATO.
14. ENVIA_LCD_DATO: esta rutina es muy similar a ENVIA_LCD_CMD, con la diferencia de que en este caso se envía una dirección y un contenido a mostrar en la pantalla o un carácter a generar en la CGRAM. El valor a enviar se carga previamente en el registro W.
15. GENERA_CHAR_CGRAM: esta rutina es la más compleja. Se utiliza para crear un nuevo carácter en la CGRAM. En total son 8 caracteres diferentes que se pueden crear, y cada carácter se compone de 8 bytes. El primer carácter se identifica con el valor 0, el segundo con el 1, y así hasta el octavo, con 7. Estas cifras se escriben en la memoria DDRAM para que se muestre el nuevo carácter. En esta rutina es necesario especificar los 8 bytes a crear y el número de identificador (que se especifica por su dirección en bytes, es decir, el primer carácter representa la posición 0 de la CGRAM, el segundo la posición 8, el tercero la posición 16, y así hasta el octavo, que se encuentra en la posición 56), el cual se carga en el registro dir_CGRAM. Además, es necesario definir los 8 bytes del carácter en una tabla y especificar la primera dirección en el puntero de tabla TBLPTR. Una vez hecho esto, ya es posible invocar a la rutina GENERA_CHAR_CGRAM.

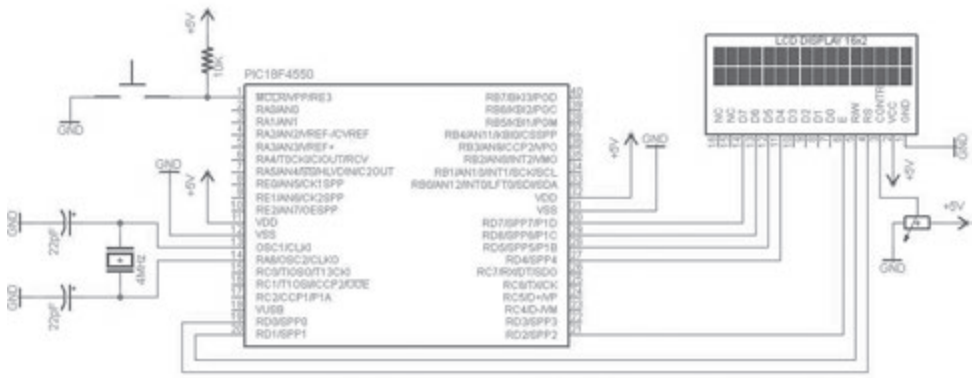
En el diagrama de conexiones del módulo LCD alfanumérico al PIC18F4550 que se muestra en el gráfico 7.12 se puede ver que el bus de control se encuentra conectado a las líneas RD0 (RS), RD1 (R/W) y RD2 (E) del Puerto D. Mientras, el bus de datos se conecta a las líneas RD4, RD5, RD6 y RD7 del mismo puerto. El cristal externo conectado al PIC18F4550 es de 4 MHz. Para este ejemplo no se utiliza el PLL interno. Sin embargo, de hacer uso del multiplicador de reloj, solo habría que actualizar las rutinas de retardo de tiempo para que se ajusten al nuevo ciclo de instrucción. Este esquema de conexión será usado para todos los ejemplos que involucren al módulo LCD en todos los capítulos restantes de este libro.

Ejemplo 7.1

A continuación se va a crear un programa para mostrar en la pantalla LCD alfanumérica el mensaje "Hola Mundo", el cual comenzará en la cuarta columna de la primera fila. Luego, el mensaje "PIC18F4550" aparecerá en la cuarta columna y en la segunda fila. Finalmente, el cursor permanecerá parpadeando. El formato del mensaje se muestra en el gráfico 7.13.

La solución se muestra en el programa 7.3. El algoritmo comienza creando una tabla con el mensaje "Hola MundoPIC18F4550". El programa carga en el puntero de tabla la dirección del mensaje y, conforme se va capturando cada carácter en el registro W, se imprime cada dígito. Cuando se detecta la lectura del carácter ASCII 'P' se coloca el cursor en la segunda fila y cuarta columna. Cabe mencionar que la primera columna se especifica cargando el valor 0 al registro W e invocando a la rutina POS_CUR_FIL2.

Gráfico 7.12. Diagrama de tiempos para la escritura de un comando o dirección en el módulo LCD haciendo uso de un bus de datos de 4 bits



Elaboración propia.

Gráfico 7.13. Mensaje en la pantalla LCD del programa 7.3

Programa 7.3. Generación de un mensaje alfanumérico en el módulo LCD

```
#include <P18F4550.INC>
#include "LCD_LIB.asm"
 cblock 0x10
        aux
```

```
 endc
 ORG 0x0000
 goto MAIN
 ORG 0x7000
 MENSAJE: da "Hola MundoPIC18F4550"
 ORG 0x0020
MAIN:
        clrf  TRISD,0;
        call  DELAY15MSEG
        call  LCD_CONFIG
        call  CURSOR_ON
        movlw UPPER MENSAJE
        movwf TBLPTRU
        movlw HIGH MENSAJE
        movwf TBLPTRH
        movlw LOW MENSAJE
        movwf TBLPTRL
        movlw .3
        call  POS_CUR_FIL1
INICIO1:
        TBLRD*+;
        movf  TABLAT,W
        movwf aux
        movlw 'P'
        subwf aux,W
        btfsc STATUS,Z
        call  CAMBIA_LINEA
        movlw 0xFF
        subwf aux,W
        btfsc STATUS,Z
        goto  FIN
        movf aux,W
        call  ENVIA_CHAR
        goto  INICIO1
CAMBIA_LINEA:
        movlw .3
        call  POS_CUR_FIL2
        return
FIN:
        goto  FIN
        END
```

Ejemplo 7.2

En el presente ejemplo se mostrará el uso de la rutina de generación de caracteres en la memoria CGRAM. Para esto se utilizarán los bytes mostrados en el gráfico 7.10. Se creará el carácter Δ, el cual no existe como carácter predefinido dentro de la memoria CGROM, y se generará el mensaje en la pantalla LCD que se muestra en el gráfico 7.14.

En el programa 7.4 se muestra la solución al ejemplo 7.2. Al igual que en el programa anterior, se crea una tabla con los 8 bytes que dan forma al carácter Δ. La dirección de esta tabla es asignada al puntero de tabla TBLPTR y, luego, se selecciona el dígito 0 como dirección base, de manera que, para mostrar el nuevo carácter en la pantalla, se cargue el valor 0x00 en el registro W y se invoque a la rutina ESCRIBE_CHAR.

Gráfico 7.14. Mensaje en la pantalla LCD del programa 7.4

Elaboración propia.

Programa 7.4. Creación de un nuevo carácter en la memoria CGRAM y presentación en la pantalla

```
#include <P18F4550.INC>
#include "LCD_LIB.asm"
cblock 0x20
        aux
        endc
ORG 0x0000
goto MAIN
ORG 0x7000
DELTA: db 0x04,0x0A,0x11,0x11,0x11,0x11,0x11,0x1F
ORG 0x0020
MAIN:
        clrf  TRISD,0;
        call  DELAY15MSEG
        call  LCD_CONFIG
        call  CURSOR_ON
INICIO:
        movlw UPPER DELTA
        movwf TBLPTRU
        movlw HIGH  DELTA
```

```
        movwf TBLPTRH
        movlw LOW   DELTA
        movwf TBLPTRL
        movlw .0
        movwf dir_CGRAM
        call  GENERA_CHAR_CGRAM
        movlw .3
        call  POS_CUR_FIL1
        movlw .0
        call  ENVIA_CHAR
        movlw 't'
        call  ENVIA_CHAR
        movlw '='
        call  ENVIA_CHAR
        movlw '5'
        call  ENVIA_CHAR
FIN:
        goto  FIN
 END
```

Ejemplo 7.3

A continuación se presenta un ejemplo de un programa más completo, en el cual se relaciona la rutina de detección de teclado con la presentación de caracteres en la pantalla LCD. El teclado matricial se conecta al Puerto B bajo la misma configuración vista en el gráfico 7.3, mientras que el LCD se mantiene conectado al Puerto D en la misma configuración presentada en el gráfico 7.12. En el presente programa se solicita el ingreso de una clave numérica a través de la pantalla LCD. Inicialmente, aparecerá el mensaje "Clave:" en la pantalla LCD alfanumérica, y el cursor estará oscilando al lado del mensaje. Conforme se ingresen dígitos por el teclado, se mostrarán «*» en la pantalla, mientras que el verdadero dígito ingresado será almacenado en la memoria de datos para su posterior validación. Este esquema se muestra en el gráfico 7.15. Cuando el usuario pulse la tecla D, entonces el programa evaluará la clave ingresada (sin tomar en cuenta la tecla D para la validación).

Gráfico 7.15. Ventana de solicitud de ingreso de clave numérica

Elaboración propia.

En el proceso de validación se comparará la clave ingresada con los dígitos ASCII '1', '2', '3' y '4'. De ser incorrecta la secuencia, en la segunda fila de la pantalla LCD se mostrará el mensaje «MAL», tal como se muestra en el gráfico 7.16, lo que indica que la clave fue errónea.

En el caso de que el proceso de validación sea satisfactorio, se mostrará el mensaje «OK» en la segunda fila, tal como se muestra en el gráfico 7.17. Una vez visto el mensaje resultante, el programa entrará en un bucle indefinido. Para repetir el proceso, el usuario tendrá que pulsar el botón de Reset para reiniciar el programa.

Gráfico 7.16. Mensaje de ingreso erróneo de la clave numérica

Elaboración propia.

Gráfico 7.17. Mensaje de ingreso correcto de la clave numérica

Elaboración propia.

En el programa 7.5 se muestra la solución de la propuesta. Se utiliza la rutina de interrupción por cambio de estado de los pines RB4-RB7, tal como la mostrada en el programa 7.1 (rutina RUT_INT). De igual manera, se utilizan las rutinas de manejo de la pantalla LCD que se encuentran en el archivo LCD_LIB.asm. Conforme se ingresan los dígitos de la clave, estos se almacenan en las direcciones 0x30, 0x31, 0x32, etc., de la memoria de datos a través de direccionamiento indirecto. Posteriormente, una vez detectada la tecla 'D', se leen estas direcciones de la memoria de datos y se comparan en orden con la secuencia de la clave. Si uno de los dígitos no coincide inmediatamente, se envía el mensaje del gráfico 7.16. De ser satisfactorias las cuatro comparaciones, se despliega el mensaje del gráfico 7.17.

Programa 7.5. Algoritmo de interacción del teclado matricial y la pantalla LCD alfanumérica para el ingreso de una clave de cuatro dígitos

```
#include <P18F4550.INC>
#include "LCD_LIB.asm"
ORG 0x00
        goto MAIN
ORG 0x08
goto RUT_INT
ORG 0x018
```

```
                retfie
ORG 0x7000
MENSAJE:   da "Clave:"
ORG 0x020
MAIN:
        clrf  LATD
        clrf  TRISD;
        clrf  tecla
        call  DELAY15MSEG
        call  LCD_CONFIG
        call  CURSOR_ON
        call  BORRAR_LCD
        movlw UPPER MENSAJE
        movwf TBLPTRU
        movlw HIGH MENSAJE
        movwf TBLPTRH
        movlw LOW MENSAJE
        movwf TBLPTRL
        movlw 0xF0
        movwf TRISB
        bcf   RCON,IPEN
        bcf   INTCON2,RBPU
        clrf  LATB
        bsf  INTCON,GIE
        bsf  INTCON,PEIE
        bsf  INTCON,RBIE
        lfsr  FSR0,0x30
INICIO:
        TBLRD*+;
        movf  TABLAT,W
        movwf var
        movlw 0xFF
        subwf var,W
        btfsc STATUS,Z
        goto  ESPERA
        movf var,W
        call  ENVIA_CHAR
        goto  INICIO
ESPERA:
        movlw .0
        xorwf tecla,W
        btfsc STATUS,Z
goto  ESPERA
        movlw 'D'
        xorwf tecla,W
```

```
        btfsc STATUS,Z
        goto  EVALUA
        movf tecla,W
        movwf INDF0
        incf  FSR0L
        movlw '*'
        call  ENVIA_CHAR
        clrf  tecla
        goto  ESPERA
EVALUA:
;Clave correcta 1,2,3,4
        lfsr  FSR0,0x30
        movf  INDF0,W
        movwf var
        movlw '1'
        cpfseq var
        goto  CLAVE_MAL
        incf  FSR0L
        movf  INDF0,W
        movwf var
        movlw '2'
        cpfseq var
        goto  CLAVE_MAL
        incf  FSR0L
        movf  INDF0,W
        movwf var
        movlw '3'
        cpfseq var
        goto  CLAVE_MAL
        incf  FSR0L
        movf  INDF0,W
        movwf var
        movlw '4'
        cpfseq var
        goto  CLAVE_MAL
        goto  CLAVE_BIEN
CLAVE_MAL:
        movlw .4
        call POS_CUR_FIL2
        movlw 'M'
        call ENVIA_CHAR
        movlw 'A'
        call ENVIA_CHAR
        movlw 'L'
```

```
            call ENVIA_CHAR
            goto FIN
CLAVE_BIEN:
            movlw .4
            call POS_CUR_FIL2
            movlw 'O'
            call ENVIA_CHAR
            movlw 'K'
            call ENVIA_CHAR
            goto FIN
FIN:
            goto FIN
            END
```

Video

Ingreso de una contraseña mediante teclado matricial y lectura de los datos a través del LCD alfanumérico.

http://bit.ly/1EOwQLU

Resolución de videos en HD

Preguntas de repaso

1. Si el registro TRISB = 0xA0, ¿cuáles son los pines del Puerto B configurados como entrada digital?
2. ¿Es posible conectar una carga de 100 Ω y generar una corriente de 50 mA a uno de los pines del PIC18F4550 configurado como salida?
3. ¿Cuál es el beneficio de implementar la rutina de detección de teclas pulsadas haciendo uso de la interrupción de cambio de estado del Puerto B?
4. ¿Cuál es la función del cursor en una pantalla LCD alfanumérica?
5. ¿Cuál es la desventaja que conlleva el uso de un bus de comunicaciones de 4 bits para realizar el control del módulo LCD alfanumérico?
6. Indique cuál es la función de los siguientes elementos en el módulo LCD alfanumérico:
   a. Memoria DDRAM
   b. HD44780
   c. Memoria CGRAM
   d. Memoria CGROM

Respuestas

Respuestas del capítulo 7:

http://bit.ly/1OvaRJP

Capítulo 8. El lenguaje ANSI C

Hace algunos años, los microcontroladores tenían los recursos muy limitados. Es decir, su memoria de programa, memoria de datos y velocidad de ejecución de instrucciones era reducida. De esta manera, el lenguaje ensamblador se convertía en la única alternativa para que el programador pueda tener el control completo de los recursos del microcontrolador y al mismo tiempo pudiera optimizar los recursos del sistema embebido, a fin de que su aplicación pueda ser ejecutada con una eficiencia máxima.

Cuando aparece el lenguaje ANSI C para sistemas embebidos, muchos programadores se mostraron reacios a utilizar dicha herramienta. El hecho de que el microcontrolador pudiera ser programado por un lenguaje de alto nivel que enmascarara el código en ensamblador resultaba aterrador, ya que el programador percibía que iba a perder el control de la optimización de recursos del microcontrolador e iba a terminar construyendo un programa totalmente ineficiente por el simple hecho de utilizar un lenguaje más «amigable».

Hoy en día los sistemas embebidos están diseñados con altas capacidades de memoria de programa y datos. Además, las empresas desarrolladoras de microcontroladores están trabajando arduamente en las mejoras de sus compiladores para lograr obtener una mayor optimización del código generado por el compilador. Por esta razón, el lenguaje ANSI C se ha convertido en el preferido por la mayoría de programadores de sistemas embebidos que ahora cuentan con un lenguaje eficiente que trae el beneficio de acelerar el tiempo de implementación de algoritmos de mediana y alta complejidad.

Microchip ofrece un compilador de licencia gratuita y la opción de utilizar el compilador de máxima eficiencia (versión PRO) por 60 días, de nombre MPLAB XC8 (descargable directamente de su página web). Esta herramienta es un Plug In que se adapta al entorno MPLAB X IDE y permite la edición y compilación de proyectos en lenguaje ANSI C. Microchip viene trabajando por varios años en este *software* y hoy en día es bastante popular en el mundo de los programadores de microcontroladores PIC18F.

8.1 Historia del lenguaje ANSI C

El lenguaje C fue creado en los Laboratorios Bell de AT & T en el año 1972 por los ingenieros Brian Kernighan y Dennis Ritchie con el objetivo de poder encontrar una forma de reescribir el sistema operativo UNIX, a fin de mejorar su portabilidad. Gracias a la eficacia que obtuvo este lenguaje, el ensamblador fue prácticamente inutilizado en la descripción del UNIX. Una de las grandes características del lenguaje C, que lo hizo altamente aceptado por los programadores de su época (y más aún en la actualidad), es que este lenguaje, aunque es de alto nivel, presenta algunas características que lo hacen parecer de bajo nivel, permitiendo estar en ciertas circunstancias al rango de un lenguaje de máquina. Es por esta razón que Kernighan y Ritchie publicaron el libro *The C Programming Language* en el año 1978. Esta publicación dio pie a la aparición de diversos compiladores en lenguaje C que abundaban en

el mercado a mediados de la década de 1980. Sin embargo, este lenguaje no era un estándar, por lo que cada nuevo compilador creado le iba agregando algunos cambios a la propuesta original de los autores.

Recién en el año 1990, el lenguaje C se convirtió en un estándar denominado ANSI C (ANSI por las siglas *American National Standards Institute*), y obviamente fue una versión mejorada de la propuesta de Kernighan y Ritchie de 1978. Posteriormente, en el año 1999, aparece la última estandarización del lenguaje ANSI C, que agregó algunos componentes del lenguaje C++. Cabe resaltar que no todos los compiladores actuales son compatibles con este estándar[40].

El ANSI C nació inicialmente para ser operado en un ambiente basado en un sistema operativo, como son las computadoras convencionales. Posteriormente, el lenguaje se empezó a utilizar en ambientes para sistemas embebidos, donde el código se genera y se compila en un *Host* (computador) y posteriormente es descargado en un *Target* (sistema embebido), haciendo uso de un programador o depurador de por medio.

En el caso específico del ambiente en sistemas embebidos, se debe destacar que ningún lenguaje ANSI C es 100% compatible con el lenguaje para otra familia o marca de microcontroladores. Al variar la arquitectura del sistema embebido, aparecen algunas variantes del lenguaje en cuanto al acceso a periféricos, manejo de interrupciones, directivas y cabeceras de configuración, que son generadas por los desarrolladores del compilador. Fuera de esto, las instrucciones que conforman la secuencia del programa tienden a ser entendibles por cualquier conocedor del lenguaje.

El compilador en C transforma el código fuente (archivo de extensión .c) en un formato objeto (archivo de extensión .o) que puede ser usado por el enlazador para producir los archivos de salida, entre ellos el de extensión .hex, que es utilizado por el programador para instalarlo en la memoria de programa del sistema embebido. Para lograr este proceso, el compilador transforma el código en ANSI C a un equivalente al lenguaje ensamblador usado por el sistema embebido. Este proceso de conversión de código se puede observar en una ventana de Desensamblaje (*Disassembly*). Los modos de optimización del compilador permiten que el lenguaje ensamblador equivalente generado sea lo más eficiente posible. Claro está que este grado de eficiencia siempre presenta un costo de adquisición.

El compilador MPLAB XC8 es una herramienta compatible con el estándar ANSI C 1999 y se enlaza directamente con el entorno MPLAB X IDE y con las herramientas que este *software* maneja. El *software* se puede obtener de forma gratuita con la opción de activación de la versión PRO (que es la que tiene una máxima eficiencia de conversión del 100%) con una licencia de 60 días que permite aprovechar los grados de optimización durante ese tiempo. Posteriormente, el compilador no optimizará más el código generado.

Hoy en día el lenguaje ANSI C es prácticamente la primera opción a utilizar por la gran comunidad de programadores de sistemas embebidos en el mundo. Las ventajas comparativas del lenguaje ANSI C para sistemas embebidos son las siguientes:

1. Facilita la generación de código de mediana y alta complejidad

Muchas veces, los desarrolladores se encuentran ante la necesidad de implementar algún tipo de algoritmo de procesamiento digital de señales, un proceso matemático complejo o un proceso de control PID en un microcontrolador. Estas tareas son muy simples de realizar en un simulador de *software*,

40 Cfr. Galeano 2009.

tales como el Matlab o LabView. Pero, cuando se desea integrar el mismo algoritmo en un sistema embebido, es preferible que este se realice en un lenguaje de alto nivel para facilitar la rapidez de la implementación y así poder tener el producto finalizado en el menor tiempo posible.

2. El código es interpretable por cualquier programador de lenguaje ANSI C y con conocimientos en sistemas embebidos

Se tiene a un experto programador en el lenguaje ensamblador del PIC18F que desea implementar un algoritmo que ejecuta la secuencia CRC (Comprobación de redundancia cíclica) sobre un conjunto de bits que recibe por uno de los periféricos del microcontrolador. Recibe un código que ya fue elaborado por otro programador de sistemas embebidos que utilizó un microcontrolador Freescale modelo MC9S12NE64 y el lenguaje ensamblador para la familia MC9S12. Dado que no conoce este lenguaje, primero tendría que estudiarlo para que, finalmente, pueda analizar el código que el programador de Freescale ha elaborado. Si este programador hubiera generado su código en lenguaje ANSI C, no sería necesario estudiar el lenguaje ensamblador de la familia de microcontroladores MCS12 para entender el programa y transcribirlo al PIC18F. En resumen, el lenguaje ANSI C facilita la interpretación del código entre distintos programadores.

3. Es mucho más fácil desarrollar aplicaciones trabajando en equipo

Cuando un proyecto se inicia, normalmente se trabaja entre al menos dos ingenieros programadores. Una vez que cada ingeniero culmina su parte del proyecto, es necesario juntar los códigos para ensamblar el proyecto completo. Cuando este proceso se realiza, habiéndose desarrollado el código en lenguaje ensamblador, la fusión de los programas resulta un dolor de cabeza, ya que se tiene que analizar en qué posición del programa debe ir guardada cada sentencia y sincronizar las diferentes rutinas. En cambio, en lenguaje C este proceso es mucho más simple, sobre todo si dos o más programadores comparten el mismo conjunto de librerías.

4. Es posible crear librerías y compartirlas con una comunidad de programadores

Siguiendo el ejemplo anterior del código CRC, el programador desarrolla su propio algoritmo para analizar una secuencia de 1200 bits y desea ponerlo en internet para que muchos programadores en el mundo puedan conocerlo y utilizarlo. Para este fin, los programadores simplemente requieren descargar la librería de su portal web, incluir el archivo en el proyecto y utilizar las funciones o macros que haya creado para darles uso. Estando el programa en una librería, se facilita su utilización para otros programadores, lo cual es una ventaja no solo para el trabajo en equipo sino también para la misma organización de un proyecto de diseño de *firmware*.

5. Si es necesario, se pueden incorporar bloques de ensamblador en el lenguaje ANSI C

¿Qué sucede en el caso de que un programador necesite optimizar una rutina al máximo y, para tal situación, requiera hacer uso del lenguaje ensamblador de todas formas? Pues bien, en este caso, el lenguaje ANSI C permite la incorporación de bloques de lenguaje ensamblador, lo que deja que ciertas partes del programa se ejecuten exactamente como el programador lo decide. El compilador no opti-

miza ni traduce aquellas sentencias escritas en el lenguaje ensamblador. Por otro lado, la sintaxis para incorporar el lenguaje de bajo nivel en ANSI C varía de acuerdo con el tipo de compilador. En MPLAB XC8 se utilizan las directivas #asm y #endasm.

6. El código en lenguaje ensamblador generado por el compilador es más eficiente que un código creado por un buen programador

Antiguamente, un buen programador de lenguaje ensamblador criticaba a los compiladores de ANSI C en el sentido de que el código máquina generado no era lo suficientemente eficiente comparado con el que el propio programador podría desarrollar. Hoy en día, esto es prácticamente una falacia. Los compiladores son muy eficientes y utilizan una característica en particular: suelen emplear todo el conjunto de instrucciones para realizar un programa equivalente en ANSI C. Esto obviamente es un aspecto que supera a la mayoría de los programadores en ensamblador que generalmente presentan la tendencia a hacer uso con mayor frecuencia de ciertas instrucciones del lenguaje sobre otras.

Por otra parte, no todo es positivo, el uso del lenguaje ANSI C para sistemas embebidos también presenta ciertas desventajas, que a continuación se enumeran:

1. El código en C siempre genera un gasto mayor en espacio de memoria de datos y programa

El lenguaje ANSI C, al ser de alto nivel, requiere que ciertos espacios de memoria estén separados para determinados tipos de variables y manejo de la pila. Esto requiere el uso de un código de inicialización, el cual es transparente para el programador en C, pero existe a nivel de lenguaje ensamblador. Este código, por supuesto, genera mayor gasto de memoria de programa y utiliza recursos de la memoria de datos. Por esta razón, no se recomienda el uso del ANSI C si el sistema embebido tiene los recursos de memoria muy limitados.

2. El compilador no es gratuito

El compilador MPLAB XC8, al igual que muchos otros compiladores de ANSI C para sistemas embebidos, se puede descargar gratuitamente de internet, pero, realmente, no es gratuito. Lo que se descarga es una versión *free* cuyo grado de optimización de código es limitado. Existen otras versiones del compilador, como la Standard y la PRO, que presentan mejores grados de optimización, permitiendo que el código generado pueda ocupar menos espacio en memoria y se ejecute más rápidamente. El uso de la versión gratuita puede dar pie a generar un código equivalente que muchas veces puede resultar ineficiente para la aplicación buscada. Este problema no existe si se programa en lenguaje ensamblador, ya que los ensambladores sí son gratuitos y no presentan más limitaciones que las dificultades propias del programador para plasmar sus ideas en un programa.

3. Es posible programar en lenguaje ANSI C sin conocer de manera completa la arquitectura del sistema embebido

Esto es bastante cierto. Cuando el ANSI C apareció como opción de programación de sistemas embebidos, la comunidad de programadores creció. Esto permitió que ingenieros de múltiples disciplinas puedan empezar a programar los sistemas embebidos. ¿Pero realmente conocen la arquitectura del microcontrolador cuando crean aplicaciones en este? En muchas situaciones se ha visto que esto no siempre es cierto: el programador puede ignorar algunos o muchos detalles de la arquitectura, ya que el lenguaje lo abstrae de estos aspectos y aun así permite generar código funcional. El problema se da cuando ciertas rutinas no operan por desconocimiento de detalles tan simples como no conocer de la existencia de los modos de activación de las resistencias de *Pull Up* internas de un puerto o no conocer el factor de multiplicación del PLL interno. En caso de hacer uso del lenguaje ensamblador, es más difícil que el programador desconozca estos aspectos[41].

8.2 La estructura de un programa en lenguaje C

La estructura de un programa en ANSI C para el MPLAB XC8 cuenta con las siguientes partes básicas:

1. Directiva de inclusión del archivo de cabecera, la cual permite adherir al programa un conjunto de funciones y macros muy útiles para configurar al microcontrolador, como la declaración de sus registros internos y bits de configuración, algunas funciones para manejo de cadenas de caracteres y manejo de ciertos periféricos, etc. La directiva de inclusión debe estar en la primera línea de texto del programa, como se muestra a continuación:

```
#include <xc.h>
```

2. Luego, se deben especificar la configuración de los fusibles o bits de configuración de operación del microcontrolador. Esta etapa es fundamental para el correcto funcionamiento del programa en el sistema embebido, ya que si se ignora, el compilador utilizará las opciones o configuraciones por defecto que, por obvias razones, pueden diferir de las opciones reales esperadas en el *hardware*. Estas opciones son diversas: el modo de reloj del CPU, el Prescaler del PLL, el Postscaler del PLL, habilitación del Perro guardián, etc. A continuación se presentan los modos básicos usados por los programas de ejemplo:

41 Cfr. Galeano 2009, revisado y modificado por Salas 2014.

```
#pragma config FOSC      = XTPLL_XT  //Cristal externo menor o igual a 4MHz.
#pragma config PLLDIV    = 1                 //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV    = OSC1_PLL2//Divisor del Postscaler = 96MHz/2 = 48MHz.
#pragma config PWRT      = ON        //Power Up Timer activo
#pragma config WDT       = OFF       //Perro guardián desactivado
#pragma config LVP       = OFF       //Low Voltage ICSP desactivado
#pragma config BOR       = OFF       //Brown Out Reset desactivado
#pragma config DEBUG     = ON        //Modo de depuración activado        .
#pragma config VREGEN = OFF       //Regulador de voltaje del USB desactivado.
#pragma config PBADEN = OFF       //Pines del Puerto B como E/S digitales.
#pragma config MCLRE     = ON        //Pin RA6 opera como Reset.
```

3. Finalmente, se debe colocar la estructura de la función main(). Dentro de esta función se ubica el código del programa, el cual estará delimitado entre las llaves { }. Cabe resaltar que en todo el proyecto solo puede existir una única función main():

```
        void main()
{
//El código del programa se ubica entre las llaves.
}
```

Para colocar un comentario en lenguaje ANSI C se utiliza el operador // si es que se desea realizar un comentario en una sola línea. En el caso de que se quiera realizar un comentario que abarque varias líneas se utiliza el operador /* para abrir el comentario y */ para cerrarlo. Por ejemplo, en el programa 8.1 se muestra un ejemplo de ambos tipos de comentario.

Programa 8.1. Ejemplo de comentarios en ANSI C

```
#include <xc.h>
//Las sentencias debajo son comentarios de una sola línea
#pragma config FOSC      = XTPLL_XT     //Uso de un cristal de 4MHz con PLL activo.
#pragma config PLLDIV    = 1            //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV    = OSC1_PLL2    //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT      = ON           //Power Up Timer activado.
#pragma config WDT       =  OFF         //Watch Dog Timer desactivado.
#pragma config LVP       =  OFF         //Low Voltage ICSP desactivado.
#pragma config BOR       =  OFF         //Brown Out Reset desactivado.
#pragma config DEBUG     =  ON          //Background Debugger activado
#pragma config VREGEN = OFF          //Regulador de voltaje del USB desactivado.
```

```
#pragma config PBADEN = OFF  //Pines de entrada analógica en modo digital.
#pragma config MCLRE  =  ON    //Habilitación del reset externo.
/*Este es un comentario
que abarca más de una línea de código*/
void main()
{
    TRISB = 0;
    LATB = 5;
    while(1);
}
```

Para efecto de los ejemplos que se presentarán, se ignorarán los bits de configuración, para evitar la extensión del código.

En el lenguaje ANSI C se trabajan básicamente con tres tipos de base numérica. La base decimal se especifica simplemente colocando la cifra numérica tal y como se escribe. Por ejemplo, a = 10 implica que la variable a toma el valor de diez en decimal. En base hexadecimal la igualdad sería a = 0x0A, mientras que en base binaria sería a = 0b00001010.

8.3 Los tipos de datos

En el lenguaje ANSI C se trabaja directamente con variables. Una variable es un alias que se asigna a una dirección en memoria. La ventaja que ofrece el ANSI C es que podemos crear la variable del tamaño que se quiera y no necesariamente tiene que estar restringida a un ancho de 8 bits. La declaración de una variable sigue la siguiente sintaxis:

```
tipo      nombre de la variable = valor inicial;
```

Los diferentes tipos de variable que permite manejar el compilador MPLAB XC8 se muestran en el gráfico 8.1. Cada tipo representa una longitud de datos específica y suelen ser longitudes múltiplos de 8 bits. Sin embargo, más adelante se verá que es posible crear tipos de datos de longitudes variables.

Gráfico 8.1. Tipos de datos del lenguaje ANSI C

Tipo	Tamaño	Valor mínimo	Valor máximo
char	8 bits	0	255
signed char	8 bits	-128	127
unsigned char	8 bits	0	255
signed int	16 bits	-32768	32769
unsigned int	16 bits	0	65535

Tipo	Tamaño	Valor mínimo	Valor máximo
signed short	16 bits	-32768	32767
unsigned short	16 bits	0	65535
signed short long	24 bits	-8,388,608	8,388,607
unsigned short long	24 bits	0	16,777,215
signed long	32 bits	-2,147,483,648	2,147,483,647
unsigned long	32 bits	0	4,294,967,295
float	32 bits	$-3.4x10^{-38}$	$+3.4x10^{38}$
double	32 bits	$-3.4x10^{-38}$	$+3.4x10^{-38}$

Elaboración propia.

Como se puede observar, a los tipos enteros se les asocia los modificadores *signed* o *unsigned*. Esto implica trabajar la variable con signo o sin signo, respectivamente. Si se ignoran estos modificadores, la variable se considera sin signo. En el formato *signed* el bit más significativo se considera el bit de signo y toda la cifra se interpreta en formato complemento a 2 por el compilador. El uso de este tipo de variable genera un poco más de carga computacional en ciertas operaciones, así que es recomendable evitar su utilización en caso de que no se requiera trabajar valores con signo.

Los tipos de 16, 24 y 32 bits ocupan 2, 3 y 4 registros de memoria de datos, respectivamente, por cada variable declarada. Si bien es cierto que facilitan la manipulación de cifras de tamaños más extensos, también generan una mayor carga computacional, ya que en el lenguaje ensamblador las instrucciones solo manipulan variables de 8 bits en cada instrucción.

Una mención especial merecen los tipos *float* y *double*. Estos tipos de datos manejan el formato de Coma flotante presentado en el capítulo 1. En el caso especial del compilador MPLAB XC8, el formato que utilizan los tipos *double* y *float* es una variante denominada formato IEEE 754, en el cual el byte menos significativo representa el exponente, el segundo byte presenta el bit de signo con 7 bits de parte decimal y los restantes dos bytes representan el resto de la cifra decimal, tal como se muestra en el gráfico 8.2.

Gráfico 8.2. Formato de punto flotante utilizado por MPLAB XC8

Signo	Exponente	Mantisa
s	$e_7e_6e_5e_4e_3e_2e_1e_0$	ddd dddddddd dddddddd dddd
s: bit de signo d: mantisa e: exponente		

Elaboración propia.

En lo que respecta al nombre de la variable, esta puede contener caracteres alfanuméricos, pero siempre debe iniciar con un carácter alfabético o con el carácter *Underline* (_). Es recomendable que el nombre de la variable esté asociado a su función en el programa, a fin de que el programador pueda interpretar rápidamente la utilidad de la variable. Por lo general, se recomienda que la variable esté

escrita en minúsculas para evitar confundirla con las constantes generadas por las directivas #define. Además, cabe recalcar que las variables son sensibles a las minúsculas y mayúsculas, por lo cual una variable de nombre *Aux* es diferente a una declarada con el nombre *aux*.

Como parte opcional queda la sección de inicialización de la variable, en la cual se le puede asignar un valor por defecto. Simplemente se agrega el operador = y el valor de inicialización. Si esta opción se ignora, la variable por defecto queda inicializada en 0.

Por ejemplo, se observa la siguiente declaración de una variable:

```
unsigned long delta = 0x90817263;
```

Aquí se puede observar que la variable delta ocupa 4 bytes. El byte menos significativo del valor asignado a la variable delta se almacenará en la dirección más baja de la memoria de datos, tal como se observa en el gráfico 8.3. Esto se conoce como el formato "*endianness*".

Normalmente, si solo se declara la variable, como en el ejemplo mostrado, el compilador se encargará de asignar la dirección de inicio en un sector predefinido. Por ejemplo, en el gráfico 8.3 se muestra como dirección de inicio el valor 0x01 que es asignado por defecto por el compilador ¿Qué sucedería si se quisiera establecer una dirección de inicio para la variable o variables que se vayan a declarar?

Gráfico 8.3. Formato "endianness" para almacenamiento de variables de más de un byte de longitud

Dirección	0x01	0x02	0x03	0x04
Contenido	0x63	0x72	0x81	0x90

Elaboración propia.

Al igual que en el lenguaje ensamblador esto es posible, la opción @ permite definir la dirección de la memoria SRAM donde se establecerá la variable. Por ejemplo, si se realiza la siguiente declaración, se obtendrá el resultado que se muestra en el gráfico 8.4.

```
unsigned long delta @0x60;
void main()
{
        delta = 0x90817263;
```

Gráfico 8.4. Declaración de una variable en una dirección predefinida por el usuario

Dirección	0x60	0x61	0x62	0x63
Contenido	0x63	0x72	0x81	0x90

Elaboración propia.

En este ejemplo, el constructor @0x60 define que a partir del sector de memoria SRAM con dirección 0x60, este valor representa la dirección absoluta en la cual se declara la variable. Aquellas variables que no son inicializadas estarían definidas en su dirección por defecto. Como ejemplo, se presenta el siguiente código:

```
unsigned long delta @0x60;
unsigned char alfa=0xA0;
void main()
{
        delta = 0x90817263;
```

En este caso, la variable delta estaría definida a partir de la dirección 0x60, tal como se mostró en el gráfico 8.4. Sin embargo, la variable alfa estaría ocupando la dirección 0x03, ya que no se modificó su dirección, por lo cual el compilador le asignará una dirección por defecto en el banco 0. Como la declaración de alfa no especifica un sector de memoria de datos, su declaración no se encuentra inicializada y, por tanto, el valor 0x03 será la dirección por defecto asignada por el compilador.

¿Y qué pasaría si se deseara que la variable alfa estuviera inicializada en la dirección 0x300? Pues bien, para esto habría que modificar el ejemplo de la siguiente manera:

```
unsigned long delta @0x60;
unsigned char alfa @0x300;
void main()
{
    delta = 0x90817263;
    alfa=0xA0;
```

Como se observa, la variable alfa se encuentra en el banco 3 con la dirección 0x300. Durante la declaración de una variable en modo absoluto (con el constructor de dirección @) no es posible asignar un valor inicial. Para esto se tiene que realizar la asignación una vez iniciado el programa dentro de la función main().

En cuanto a la memoria de programa, también existen modificadores que permiten declarar tablas de datos o definir sectores del programa donde estará guardado el código de las instrucciones. Si se recuerdan los ejemplos vistos en lenguaje ensamblador, se hacía uso de la directiva ORG para definir una dirección de inicio del programa o de declaración de una tabla de bytes. En MPLAB XC8 se utiliza un modificador adicional: const.

Se debe recordar, del lenguaje ensamblador, que para crear una tabla de datos con un mensaje ASCII se pueden escribir las siguientes directivas:

```
ORG 0x7000
TABLA: da “Programar en ANSI C es sencillo”
```

Pues bien, el equivalente en lenguaje ANSI C sería lo siguiente:

```
const char TABLA[] @0x7000 = "Programar en ANSI  C es sencillo";
```

El modificador const especifica que el tipo de variable es constante (esto implica que requiere ser obligatoriamente inicializada y su valor no puede ser modificado durante la ejecución del programa). Además, establece que la variable debe estar almacenada en la memoria de programa y TABLA[] es el nombre del vector que será utilizado para extraer los caracteres del mensaje (el uso de vectores se verá más adelante en este capítulo). Finalmente, el mensaje inicial aparece entre comillas dobles. Cada carácter del mensaje es el código ASCII de la letra que representa. El mensaje estará grabado en la memoria de programa, tal como se observa en el gráfico 8.5.

Gráfico 8.5. Almacenamiento de los caracteres del mensaje que compone la tabla de datos

Dirección de la memoria de programa	Contenido
0x7000	'P'
0x7001	'r'
0x7002	'o'
0x7003	'g'
0x7004	'r'
0x7005	'a'
0x7006	'm'
0x7007	'a'
0x7008	'r'
0x7009	' '
0x700A	'e'
0x700B	'n'
0x700C	' '
0x700D	'A'
0x700E	'N'
0x700F	'S'
0x7010	'I'
0x7011	' '
0x7012	'C'
0x7013	' '
0x7014	'e'
0x7015	's'
0x7016	' '
0x7017	's'
0x7018	'e'
0x7019	'n'

Dirección de la memoria de programa	Contenido
0x701A	'c'
0x701B	'i'
0x701C	'l'
0x701D	'l'
0x701E	'o'

Elaboración propia.

En cuanto al programa, también es posible definir la dirección de inicio de este, para que las instrucciones en ensamblador equivalentes estén almacenadas a partir de esta dirección específica. Para esto se hace uso nuevamente del constructor de dirección @, que es equivalente a la directiva ORG <dirección> usada en el lenguaje ensamblador. Observemos el siguiente ejemplo:

```
void main() @0x300
{
        TRISB = 0x00;
        LATB = 0x56;
        while(1);
}
```

Esto generaría el siguiente código equivalente en lenguaje ensamblador:

```
  ORG 0x300
MAIN:
        clrf TRISB
        movlw 0x56
        movwf LATB
LAZO:
                bra LAZO
                END
```

8.4 Los operadores en ANSI C

Los operadores en ANSI C permiten realizar operaciones entre variables y registros de función especial. Existen tres grandes categorías de operadores:

1. Operadores aritméticos
2. Operadores de campo de bit
3. Operadores lógicos y relacionales

Los operadores aritméticos se muestran en el gráfico 8.6. En total son siete.

Gráfico 8.6. Operadores aritméticos del ANSI C

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Residuo
++	Auto incremento
--	Auto decremento

Elaboración propia.

Para observar el funcionamiento del operador de suma, se muestra un ejemplo en el programa 8.2. A continuación, se analiza el código.

Programa 8.2. Ejemplo de funcionamiento del operador de suma

```
unsigned int k;
unsigned char a,b;
void main()
{
        a=50;
        b=220;
        k = a+b;
        while(1);
}
```

Como se puede observar, se crean tres variables: *k* de 16 bits sin signo, *a* de 8 bits sin signo y *b* de 8 bits sin signo. La variable *a* se inicializa con el valor de 50, mientras que la variable *b* se inicializa con el valor 220. Ambas magnitudes pueden ser representadas con 8 bits. Posteriormente, la tercera instrucción $k = a + b$ da como resultado 270, ya que la variable *k*, al ser de 16 bits, puede albergar un resultado que rebasa los 8 bits de dimensión (270 es la suma de dos variables de 8 bits, de las cuales ninguna podría albergar dicho resultado).

Por otro lado, el operador de resta permite realizar la sustracción de dos variables o registros del microcontrolador. La sustracción respeta el formato complemento a 2. En el programa 8.3 se muestra un ejemplo donde la variable *k* asume el valor de la resta 50-80 que toma el valor -30 en formato complemento a 2 que se representa con el valor 0xFFE2. Sin embargo, se debe tener cuidado con el manejo de las cifras negativas en formato decimal. Por ejemplo, si *b*= -80 (0xB0), entonces el resultado de *a*-*b* sería 0x32-0xB0 = 0xFF82, que en complemento a 2 representa -126 y no tiene relación con el resultado a nivel decimal, pero sí a nivel binario.

Programa 8.3. Ejemplo de resta de variables

```
int k;
char a,b,c;
void main()
{
        a=50;
        b=80;
        k = a-b;
        while(1);
}
```

El proceso de multiplicación con el compilador MPLAB XC8 hace uso intensivo de la instrucción mulwf para realizar esta operación. Al igual que la sumatoria, el operador de multiplicación debe tomar en cuenta el resultado de la operación. Observemos el ejemplo del programa 8.4.

Programa 8.4. Ejemplo de multiplicación de variables

```
int k;
unsigned char a,b;
void main()
{
        a=50;
        b=30;
        k = a*b;
        while(1);
}
```

La variable *a* se inicializa con el valor de 50 y *b* con 30. El resultado de la multiplicación es 1500 o 0x5DC.

El lenguaje ANSI C posee operadores de división para obtener el cociente (/) y residuo (%) de esta operación. Como se vio en el capítulo 5, el lenguaje ensamblador del PIC18F no contiene una instrucción de división, con lo cual para implementar estas operaciones se requiere de una rutina completa en ensamblador que obviamente presenta un tiempo de ejecución muy superior a las operaciones de suma, resta y multiplicación. Es posible también tener dividendos y divisores en formato complemento a 2. El procedimiento de división mantendrá el signo que corresponda al resultado. Para esto, observemos el programa 8.5.

Programa 8.5. Ejemplo de los operadores de división y residuo de la división

```
int k;
char a,b,c;
void main()
{
        a=50;
        b=-3;
        k = a/b;       //k = -16
        c = a%b;      //c = 2
        while(1);
}
```

En el programa 8.5 se puede observar que el dividendo o variable a toma el valor de 50, mientras que el divisor toma el valor -3. Al asignar el cociente de la división a la variable *k*, esta toma el valor -16, el cual es el correspondiente a esta operación. Por otro lado, el residuo recibido en la variable *c* es 2, ya que este no debería tener signo.

Finalmente, se tienen los operadores de auto incremento y auto decremento, los cuales permiten incrementar o disminuir en un dígito o en varios el valor de una variable. El auto incremento o auto decremento puede ser posterior o anterior. Esto significa que el valor de una variable se puede asignar a otra y luego realizar el incremento, o primero se puede realizar el incremento y luego asignar el valor a la variable. Para esto, observemos el programa 8.6.

Programa 8.6. Ejemplo de operación del auto incremento y auto decremento

```
char a,b,c;
void main()
{
        a = 5;
        b = a++;     // b = 5, a = 6
        c = --a;      // c = 5, a = 5
        while(1);
}
```

En el programa mostrado, la variable *a* se inicializa con el valor de 5. Luego, *b* toma el valor de la variable *a* (5) y, posteriormente, *a* se auto incrementa, tomando el valor de 6. A continuación, la variable *c* toma el valor de *a* luego del auto decremento, con lo cual la variable *a* es 5. Este valor es asignado a *c*.

Por otro lado, para realizar operaciones binarias en las cuales se puedan manipular los bits de cada variable, se cuenta con los operadores de campo de bit, los cuales se presentan en el gráfico 8.7.

Gráfico 8.7. Operadores de campo de bit

Operador	Operación
&	AND
|	OR
^	XOR
~	NOT
<<	Corrimiento a la izquierda
>>	Corrimiento a la derecha

Elaboración propia.

En el programa 8.7 se muestra un ejemplo de funcionamiento de los operadores AND, OR, XOR y NOT. La variable *a* se inicializa con el valor binario 10100011_2 y la variable *b* con 11000001_2. Ambas variables son sometidas a los cuatro operadores en las asignaciones siguientes. Para manipular este tipo de operadores es conveniente hacer uso de la base binaria, para poder tener una mejor visualización de los resultados de cada operación.

Programa 8.7. Ejemplo de operaciones binarias

```
char a,b,c,d,e,f;
void main()
{
        a = 0b10100011;
        b = 0b11000001;
        c = a & b;          // c = 10000001₂
        d = a | b;          // d = 11100011₂
        e = a ^ b;          // e = 01100010₂
        f = ~a;             // f  = 01011100₂
        while(1);
}
```

Los operadores de desplazamiento de bit a la derecha y a la izquierda son bastante útiles para realizar operaciones de división o conversión binario BCD o truncamiento de datos. Para observar la funcionalidad de estos operadores basta con observar el programa 8.8. Si se tiene que las variables $a = 11001111_2$ y $b = 00110000_2$ y se quiere que la variable de 16 bits $k = 0011001111001100_2$. Es decir, los 8 bits más significativos de *k* sean el valor de *a* y los 8 bits menos significativos de *k* el valor de *b* con el registro *k* corrido a la derecha dos posiciones.

Programa 8.8. Ejemplo de manejo de los operadores de desplazamiento de bit

```
char a,b;
int k;
void main()
{
        a = 0b11001111;
        b = 0b00110000;
        k = a;
        k <<=8;          //k = 11001111 00000000₂
        k |= b;          //k = 11001111 00110000₂
        k >>=2;          //k = 00110011 11001100₂
        while(1);
}
```

Cabe mencionar que la instrucción k <<= 8; es equivalente a realizar la operación k = k<<8. Igual para el caso de k |= b; equivale a realizar k = k | b y finalmente, la instrucción k>>=2; equivale a realizar k = k>>2;.

Finalmente, quedan los operadores lógicos y relacionales, que son muy útiles para el manejo de las sentencias condicionales e iterativas que se verán más adelante en este capítulo. Los operadores lógicos solamente pueden retornar dos valores: 1 (verdadero) y 0 (falso). Cualquier valor diferente de 0 en una operación lógica es considerado verdadero. Los operadores lógicos y relacionales se observan en el gráfico 8.8.

Gráfico 8.8. Operadores lógicos y relacionales

Operador	Operación
==	Igual que
!=	Diferente de
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
&&	Y
||	O
!	Negación

Elaboración propia.

Los operadores lógicos y relacionales permiten comparar valores almacenados en variables o registros internos del microcontrolador con determinados valores constantes. En el programa 8.9 se muestra un ejemplo de uso de los nueve operadores lógicos y el valor que obtienen.

Programa 8.9. Ejemplo de operación de los operadores lógicos y relacionales

```
char a= 18, b=15;
char c;
void main(void)
{
        c = a == b;       //  c = 0 (Falso)
        c = a != b;       //  c = 1 (Verdadero)
        c = a > b;        //  c = 1 (Verdadero)
        c = a < b;        //  c = 0 (Falso)
        c = a >= b;       //  c = 1 (Verdadero)
        c = a <= b;       //  c = 0 (Falso)
        c = a && b;       //  c = 1 (Verdadero)
        c = a || b;       //  c = 1 (Verdadero)
        c = !a;           //  c = 0 (Falso)
        while(1);
}
```

Cabe mencionar lo que sucede en las últimas tres instrucciones. *c* = *a* && *b* implica la operación Y entre *a* y *b*. Como se sabe, esta operación lógica se realiza con los valores 18 y 15, los cuales son considerados verdaderos por el compilador, con lo cual *c* obtiene el valor de 1. Igual ocurre para la operación *c* = *a* || *b*, en la que ambos operandos son verdaderos y por lo tanto *c* = 1. Finalmente, *c* = !*a* retorna el valor 0, ya que *a* es 18 y esto representa el valor verdadero, negando dicho valor se obtiene 0 o falso.

8.5 Sentencias condicionales e iterativas

Las sentencias condicionales e iterativas permiten controlar el flujo de ejecución de instrucciones en el programa, lo cual sirve para que el algoritmo creado pueda tomar decisiones sobre la base de parámetros de entrada. Las sentencias condicionales evalúan una condición lógica y, en función de la veracidad o falsedad de esta, deciden ejecutar un conjunto de sentencias de programa. En cambio, las sentencias iterativas repiten un procedimiento una determinada cantidad de veces hasta que una condición se haga falsa.

8.5.1 La sentencia condicional IF-ELSE

De forma general, la sentencia IF-ELSE evalúa una o varias condiciones a la vez para ejecutar un conjunto de sentencias de acuerdo con la veracidad de la o las condiciones. El formato de esta sentencia es el siguiente:

```
        if (condición 1)
        {
                BLOQUE DE INSTRUCCIONES 1
                (SI LA CONDICIÓN 1 ES VERDADERA)
        }
        else if (condición 2)
        {
                BLOQUE DE INSTRUCCIONES 2
                (SI LA CONDICIÓN 2 ES VERDADERA)
        }
        .....
        else if (condición N)
        {
                BLOQUE DE INSTRUCCIONES N
                (SI LA CONDICIÓN N ES VERDADERA)
}
else
{
        BLOQUE DE INSTRUCCIONES RESTANTES
        (SI NO ES NINGUNA DE LAS CONDICIONES ANTERIORES)
}
```

El uso de las llaves ({ }) es optativo si es que el bloque de instrucciones se encuentra conformado por una sola instrucción, mientras que es obligatorio si este bloque se compone de más de una instrucción. La sentencia condicional if empieza evaluando la primera condición. Si esta resulta verdadera, se ejecuta el primer bloque de instrucciones y la sentencia se da por terminada. De ser falsa la primera condición, se analiza la segunda condición; si esta es verdadera, se ejecuta el segundo bloque de instrucciones. Si la condición es falsa, se continúa la consulta con la tercera condición. Si todas las condiciones ubicadas dentro de la sentencias else if resultan falsas, entonces se ejecuta el bloque de instrucciones correspondiente a la sentencia else.

Por ejemplo, en el código del programa 8.10 se observa que la variable *a* se inicializa con el valor de 3. Inicialmente se comprueba la condición *a*<2. Por razones obvias, esta condición resulta falsa. Por lo tanto, la siguiente sentencia a ser ejecutada es la evaluación de la segunda condición, que corresponde a las instrucciones *a*>= 2 y *a*<4. Dado que el valor es 3 y cumple con esta condición, la variable *b* tomará el valor de 10 y la variable *c* será 3. Finalmente, se terminará el condicional if ejecutándose la instrucción while(1) de manera indefinida.

Programa 8.10. Ejemplo de la instrucción condicional if-else

```
char a,b,c;
void main()
{
        a = 3;
        if(a<2)
        {
                b = 10;
                c = 3;
        }
        else if(a>=2&& a<4)
        {
                b = 20;
                c = 7;
        }
        else if(a>=4 && a<10)
        {
                b = 30;
                c = 10;
        }
        else
        {
                b = 11;
                c = 12;
        }
        while(1);
}
```

Una pregunta interesante que nace al analizar el programa 8.10 es ¿qué hubiera pasado si la variable *a* valiera 150? Pues la respuesta es muy simple. Se hubiera analizado la primera condición, la segunda y luego la tercera, las cuales hubieran resultado falsas, para finalmente ejecutar el bloque de instrucciones correspondiente a la condición restante por defecto: *a*>=10. Entonces, la variable *b* tomaría el valor de 11 y *c* el valor 12. Lo que se debe entender de este análisis es que, dependiendo del valor que tome la variable sujeta al análisis (variable *a*), el proceso de asignación del bloque de ejecución de instrucciones puede demorar más tiempo, sobre todo si se compone de varias condiciones else if.

8.5.2 La sentencia condicional SWITCH

Un método para evitar el problema de retardo en la detección de la condición correcta dentro de las sentencias if-else es el uso de la sentencia switch. Esta sentencia condicional presenta el siguiente formato:

```
        switch (variable)
        {
                case valor 1:
                        BLOQUE DE INSTRUCCIONES 1
                        break;
                case valor 2:
                        BLOQUE DE INSTRUCCIONES 2
                        break;
                ……
                case valor N:
                        BLOQUE DE INSTRUCCIONES N
                        break;
                default:
                        BLOQUE DE INSTRUCCIONES RESTANTES
}
```

Con el uso del condicional switch se logra que una variable se someta a evaluación. Cada caso (sentencia case:) corresponde a un valor específico de la variable. Por ejemplo, si la variable en cuestión es declarada de tipo char, entonces existen 256 casos posibles, empezando desde 0 hasta 255. Este es uno de los puntos en contra que presenta esta sentencia, ya que si se quieren evaluar todos los valores posibles de la variable se necesita escribir cada condición con la sentencia case:, lo cual hace el código más extenso, pero, a la vez, más eficiente, comparado con la sentencia if. Si solo se quiere evaluar un conjunto de instrucciones, las opciones restantes pueden ser determinadas por la sentencia *default*. Por ejemplo, si se desea reescribir un código equivalente al programa 8.10 con la sentencia switch, se puede observar el código mostrado en el programa 8.11.

Programa 8.11. Ejemplo de uso de la sentencia switch

```
char a,b,c;
void main()
{
        a = 3;
        switch(a)
        {
                case 0:
                case 1:
                        b = 10;
                        c = 3;
                        break;
                case 2:
```

```
                case 3:
                        b = 20;
                        c = 7;
                        break;
                case 4:
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                        b = 30;
                        c = 10;
                        break;
                default:
                        b = 11;
                        c = 12;
        }
        while(1);
}
```

Como se observa en el programa 8.11, la sentencia switch no evalúa condiciones lógicas, sino más bien los valores que puede tomar la variable. Los dos primeros casos corresponden para $a = 0$ y $a = 1$, que son los rangos correspondientes a la condición $a< 2$. Para ambos casos se asigna el valor 10 a la variable *b* y 3 a la variable *c*. Dado que el valor de la variable *a* no es ninguno de los casos anteriores, esta condición no se ejecutará. Dado que la variable $a = 3$ se ingresa a la condición case 3: donde se ejecutan las instrucciones $b = 20$ y $c = 7$.

La instrucción break; permite culminar el condicional switch, llevando la ejecución del programa a la instrucción *while(1)*. Si no se colocara la sentencia *break*; en este punto, inmediatamente se ejecutarían las siguientes instrucciones que corresponden a $b = 30$ y $c = 10$, las cuales no corresponden.

Finalmente, la sentencia default: representa todos los posibles valores que puede tomar la variable a y que no han sido considerados en las sentencias case: anteriores. Para el ejemplo del programa 8.10, estos valores corresponden a la condición $a>=10$.

La sentencia switch es muy útil para la implementación de máquinas de estados, donde se quiere que un programa ejecute múltiples procesos en función a determinados estados que dependen de eventos externos e internos. Más adelante en este capítulo se verán ejemplos de programas que hacen uso de esta sentencia para este fin.

8.5.3 La sentencia WHILE

La sentencia iterativa *while()* permite evaluar una condición o varias condiciones conectadas a través de operadores lógicos. Si el postulado resulta verdadero, entonces el bloque de instrucciones delimi-

tado por la sentencia se ejecuta. Nuevamente, la condición se evalúa y, de mantenerse en estado de verdadero las condiciones dentro del paréntesis, las instrucciones se volverán a ejecutar. De forma general, la sentencia *while()* se representa bajo el siguiente formato:

```
while (condición)
{
        BLOQUE DE INSTRUCCIONES
}
```

En los ejemplos anteriores se hizo uso de la sentencia *while(1)*; cada vez que se culminaba un programa. Esta instrucción representa un bucle indefinido, ya que dentro del argumento se coloca la constante 1, que siempre es verdadera. Esta instrucción es equivalente a la siguiente sentencia en ensamblador: BUCLE: bra BUCLE. En el programa 8.12 se muestra un ejemplo de funcionamiento de la sentencia iterativa *while()*. Este programa resulta bastante interesante para el análisis.

Programa 8.12. Ejemplo de operación de la sentencia iterativa while

```
unsigned char a,b,c;
void main()
{
        a = 5;
        b = 2;
        c = 1;
        while(a<=5)
        {
                b++;
                c = c + b;
                a--;
        }
        while(1);
}
```

En el ejemplo mostrado, la variable *a* se inicializa con 5 y las variables *b* y *c* con 2 y 1, respectivamente. La condición de iteración es que la variable *a* sea menor o igual a 5, lo cual es verdadero. Cada vez que se ejecuta el bloque de instrucciones, la variable *a* disminuye en un dígito. Esto implica que el proceso se va a repetir para *a* = 5, 4, 3, 2, 1 y 0, que son las condiciones que cumplen con la condición *a*<= 5. Cuando la variable *a* sea 0 y se le reste un dígito, su nuevo valor será 0xFF (ya que ha sido declarada como unsigned char de 8 bits). Como 255 no es menor o igual a 5, la condición será falsa y, por lo tanto, se culminará el bucle iterativo while(*a*<=5), con lo cual se ejecutará la instrucción *while(1)*. Entonces, al finalizar el programa, la variable *b* tomará el valor de 8 y *c* de 34.

Sin embargo, ¿qué pasaría en el programa 8.12 si las variables *a*, *b* y *c* se declararan de tipo signed char? En este caso, cuando se hayan iterado los valores de la variable *a* para 5, 4, 3, 2, 1 y 0, se evaluaría la séptima condición, que es 0xFF <= 5, lo cual daría como resultado VERDADERO. Esto es obvio, ya que al declararse las variables como tipo con signo, el valor 0xFF representa el complemento a 2 de -1 y este valor sigue siendo menor o igual a 5. En esta situación, la iteración sería mucho más extensa, ya que todos los valores entre 0xFF y 0x80 resultan ser menores o iguales a 5. Recién cuando la variable *a* tome el valor 0x7F (127) la condición resultaría falsa, es decir, 134 iteraciones después.

8.5.4 La sentencia DO-WHILE

Si se analiza el programa 8.12 y se realizara el siguiente cambio al inicio del programa: *a*= 8, ¿qué pasaría? Pues la respuesta es obvia, dado que 8 no es menor o igual a 5, la condición sería falsa y la siguiente instrucción a ejecutar sería *while(1)*. Este detalle es un inconveniente del bucle *while()*. Cuando la condición es falsa desde un inicio, el conjunto de instrucciones involucrado dentro de la sentencia iterativa nunca se ejecuta. ¿Cómo podría solucionarse este problema para que al menos una vez se pueda ejecutar el bloque de instrucciones?

La respuesta a la pregunta es muy simple: haciendo uso de la sentencia iterativa do-while(). Esta sentencia es muy similar al bucle while(), con la diferencia de que la condición de iteración se evalúa al final del bloque de instrucciones y no al comienzo, lo cual permite ejecutar dicho bloque al menos una vez si es que la condición es falsa desde un inicio. El formato de la sentencia do-while() se muestra a continuación:

```
while (condición)
{
        BLOQUE DE INSTRUCCIONES
}
```

El ejemplo del programa 8.13 es una modificación al programa 8.12. Aquí se usa el bucle iterativo *do-while()* en reemplazo de while(*a*<=5) y se inicializa la variable *a* con el valor de 10. Como se sabe, este valor inicial no cumple con la condición para la iteración. Sin embargo, al ingresar al bucle do-while() la variable *b* toma el valor de 3 y *c* el valor de 4. Luego, la variable *a* disminuye en un dígito, tomando el valor de 9, el cual sigue siendo mayor que 5, por lo cual, la condición de evaluación sigue siendo falsa y se culmina el bucle, ejecutándose la sentencia while(1).

Programa 8.13. Ejemplo de operación del bucle iterativo do-while()

```
unsigned char a,b,c;
void main()
{
        a = 10;
        b = 2;
        c = 1;
        do
        {
                b++;
                c = c + b;
                a--;
        }while(a<=5);
        while(1);
}
```

8.5.5 La sentencia FOR

La sentencia iterativa *for()* permite repetir un bloque de instrucciones una cantidad determinada de veces, partiendo de una condición de inicio, y una condición de fin. Si bien es cierto que el proceso que se realiza con la sentencia *for()* se puede desarrollar de modo equivalente con el bucle *while()*, el formato de la sentencia iterativa *for()* es más ordenado y entendible por el programador para el caso de un bloque repetitivo bajo cierto número de iteraciones. Este formato se muestra a continuación:

```
for (condición inicial; condición de fin; variación de la variable sometida a la condición en
cada iteración)
{
        BLOQUE DE INSTRUCCIONES
 }
```

La condición inicial, por lo general, es un valor fijo que se le asigna a una variable para la primera iteración. La condición de fin es la condición que evalúa a la variable relacionada a la condición inicial. Si esta condición es falsa, el bloque de instrucciones se vuelve a ejecutar. Finalmente, la variación es una instrucción que altera el valor de la variable condicionada, de tal manera que, durante cada iteración, esta variable va cambiando a un valor que se aproxima a la condición falsa. Para entender mejor el funcionamiento, se analiza el programa 8.14.

Programa 8.14. Ejemplo de funcionamiento de la sentencia iterativa for()

```
unsigned char a,b,c;
void main()
{
        b = 5;
        c = 0;
        for(a = 0;a<10;a++)
        {
                c +=b;
        }
        while(1);
}
```

En el programa mostrado se puede observar que la condición de inicio es *a* = 0; la condición de fin es *a*<10 y la tasa de variación por iteración es *a++*. Esto implica que el proceso de iteración comienza con la variable *a* = 0 y por cada iteración, esta se incrementará en un dígito. Mientras la variable a se mantenga menor a 10, las iteraciones se repetirán. Una vez que esta condición no cumpla, el bucle *for()* habrá terminado.

Dado que el incremento de la variable a es en un dígito y esta variable comienza en 0, habrá diez condiciones que permitirán repetir el bloque de instrucciones esta misma cantidad de veces. Las condiciones son *a* = 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Cuando esta variable tome el valor de 10, al momento de evaluar la condición de salida se terminará el proceso. Al finalizar, la variable *c* tendrá el valor de 50 (ya que se incrementó en cinco dígitos durante 10 iteraciones) y la variable *a* mantendrá el valor de 10.

Existen otras sentencias de apoyo a los bucles iterativos que permiten forzar a la siguiente iteración (sentencia continue) o a salir del bucle, como lo es el comando *break*. Si se desea repetir el mismo proceso del programa 8.14, pero, por alguna razón, la variable de análisis fuera un número primo, las instrucciones tendrían que ignorarse y continuar con la siguiente iteración. Para esto se analiza el programa 8.15.

Programa 8.15. Ejemplo de funcionamiento de la sentencias continue y break

```
unsigned char a,b,c;
unsigned char es_primo=0;
void main()
{
        b = 5;
        c = 0;
        for(a = 0;a<10;a++)
        {
```

```
                es_primo = 1;
                for(b=2;b<a;b++)
                {
                        if(a%b == 0)
                        {
                                es_primo = 0;
                                break;
                        }
                }
                if(es_primo == 1) continue;
                c +=5;
        }
        while(1);
}
```

En el programa propuesto se considera que las cifras 0, 1, 2, 3, 5 y 7 son números primos. La variable *es_primo* presenta el valor de 1 cuando el número analizado es primo y 0 si el número no es primo (quedarían los valores 4, 6, 8 y 9). Todo número primo se caracteriza porque solo es divisible entre 1 y entre sí mismo.

La forma para detectar si un número es primo es comenzar a dividirlo entre 2 hasta una cifra un dígito menor al mismo número. Por ejemplo, para saber si la cifra 5 es primo se divide este valor entre 2 hasta 5-1 = 4. Si en todo ese rango de divisiones uno de los residuos resulta 0, entonces el número no será primo. Por lo tanto, la variable *es_primo* se actualiza con el valor de 0. La instrucción *break* finaliza el bucle *for()* interno e inmediatamente se ejecuta la instrucción *if* en donde se evalúa el valor de la variable *es_primo*. Si esta variable es igual a 1, el número no es primo y, por tanto, se ejecuta la sentencia *continue*, la cual genera la siguiente iteración, ignorando la instrucción *c+=5*. Al finalizar el programa, la variable *c* tomará el valor de 20, ya que solo se pudo incrementar cuatro veces: cuando a tomó los valores de 4, 6, 8 y 9.

Otra instrucción del ANSI C existente que permite variar el flujo de ejecución de un programa es la sentencia *goto <ETIQUETA>*. Al igual que en el lenguaje ensamblador, la sentencia goto permite llevar el PC a otra dirección de la memoria de programa direccionada por el valor de la etiqueta. De igual manera, en ANSI C se pueden establecer etiquetas en cualquier dirección de instrucción y saltar directamente a esa dirección. Por ejemplo, en el programa 8.16 se muestra el mismo algoritmo mostrado en el programa 8.15, con la diferencia de que la sentencia *break* se reemplaza con la instrucción goto SALIR.

Programa 8.16. Ejemplo de funcionamiento de la instrucción goto

```
unsigned char a,b,c;
unsigned char es_primo=0;
void main()
{
        b = 5;
        c = 0;
        for(a = 0;a<10;a++)
        {
                es_primo = 1;
                for(b=2;b<a;b++)
                {
                        if(a%b == 0)
                        {
                                es_primo = 0;
                                goto  SALIR;
                        }
                }
SALIR:
                if(es_primo == 1) continue;
                c +=5;
        }
        while(1);
}
```

8.6 Funciones y librerías

Uno de los pilares del ANSI C es el poder encapsular el código para que este pueda ser reutilizado en otros proyectos y simplifique la elaboración de nuevos programas. Este encapsulamiento consiste en la creación de funciones y librerías.

Se tiene el caso de que se desea desarrollar un nuevo proyecto para una aplicación con un microcontrolador PIC18F. En esta aplicación es de suma importancia evaluar si una variable presenta un número primo o no. Para detectar si el número es primo se requiere de una pequeña rutina basada en el bucle *for()* y en el cálculo del residuo de la división de dicho número. Este procedimiento ocupa al menos unas cuatro líneas de código. ¿Es estrictamente necesario repetirlo cada vez que se va a hallar un número primo en el programa?

La respuesta a esta pregunta es, obviamente, no. Una de las ventajas del ANSI C es la posibilidad de crear funciones. Una función es un bloque de código que se puede encapsular y que presenta entradas y salidas. Por ejemplo, para saber si un número es primo o no, se puede encapsular el código de análisis del número. La entrada a esta función sería el número que se desea analizar y la salida un valor representativo que indique si la cifra es o no prima.

Una función establece el código con las instrucciones que siempre se repiten, las entradas a este código y las salidas que deben retornarse. De forma general, una función se expresa de la siguiente manera:

```
Tipo Nombre de la función(tipo argumento 1, tipo argumento 2,…, tipo  argumento 3)
{
        BLOQUE DE INSTRUCCIONES
        return(valor de retorno)
}
```

Tipo de la función es el tipo de variable que esta retornará. Por ejemplo, un tipo puede ser unsigned char. Esto implica que la función va a retornar un valor numérico comprendido entre 0x00 y 0xFF. El nombre de la función puede ser cualquier cadena alfanumérica, la cual siempre deberá comenzar con un carácter alfabético o con un guion bajo (_). Los argumentos son la entrada de la función, y cada argumento tiene que ser declarado con un tipo de dato específico. Cada argumento puede mantener su propio tipo de dato de forma independiente.

El bloque de instrucciones se ubica entre las llaves y finalmente queda la instrucción de retorno *Return*. Dentro del paréntesis se coloca el valor a retornar, que debe ser del mismo tipo de datos del que fue declarada la función. Si no se requiere que la función retorne ningún valor, esta se puede declarar de tipo *void*, lo cual implica que la función simplemente ejecutará instrucciones, pero no habrá valor de retorno. De la misma manera, se coloca el tipo *void* en la zona de argumentos de la función para indicar que no va a recibir ningún parámetro de entrada.

Toda función debe ser declarada antes de ser definida en el programa. Para esto, previamente a su definición se debe colocar el tipo, nombre y argumentos de la función terminando con un punto y coma, tal como se declara una variable. Esto se conoce como «definición del prototipo de la función».

Por ejemplo, si se quisiera crear una función que indique si un número es primo o no, se puede realizar la siguiente declaración:

```
#define ES_PRIMO 1
#define NO_ES_PRIMO 0
unsigned char primo(unsigned char num);
```

La directiva #define se utiliza para crear un alias a una constante o inclusive a bloques de código. Esto permite mejorar la legibilidad del programa. #define ES_PRIMO 1 es equivalente en lenguaje ensamblador a usar la directiva ES_PRIMO EQU 1 o también #define ES_PRIMO 1, que tiene la misma nomenclatura. En este ejemplo, lo que se quiere dar a entender es que, si la función de nombre primo retorna el valor 1, entonces el número evaluado es primo, y si retorna 0, el número no es primo. El parámetro recibido por la función es un entero de 8 bits sin signo de nombre num, el cual será evaluado para determinar si es o no primo.

La segunda parte consiste en definir la estructura de la función, en la cual se comienza colocando el prototipo; luego, entre las llaves, se colocan las instrucciones y el valor de retorno. Para el ejemplo de la función primo se tendría la siguiente estructura:

```
unsigned char primo(unsigned char num)
{
        unsigned char i = 0;
        if(num<=3) return(ES_PRIMO);
        for(i=2;i<num;i++)
        {
                if(num%i==0) return(NO_ES_PRIMO);
        }
        return(ES_PRIMO);
}
```

Como se puede ver, dentro de la estructura de la función se declara una variable de nombre *i* que va a servir para las operaciones de iteración en el bucle for, las cuales permiten analizar el número ingresado como parámetro. Esta variable es de tipo local, ya que solo existe dentro del ámbito de la función y no puede ser accedida fuera de esta. La variable i se crea al momento de ejecutar la función y se destruye el momento de salir.

La primera operación de la función primo se encarga de evaluar si el número ingresado es menor o igual a 3. Dado que para este ejemplo se considera que las cifras 0, 1, 2 y 3 son primas, si esta condición resulta verdadera, se retornará el valor ES_PRIMO. Con esto la función quedará finalizada, ya que después de la ejecución de la instrucción *return()*, el resto de instrucciones de la función no se ejecutarán.

En caso de que el parámetro sea mayor que 3, entonces será evaluado en el bucle for. Basta que una de las divisiones dé residuo cero para que el número no sea considerado primo, por lo cual, si esta condición es verdadera tan solo una vez, se retornará el valor NO_ES_PRIMO. Finalmente, si la cifra nunca tuvo un residuo igual a cero, se saldrá del bucle *for()* y se ejecutará la última instrucción return(ES_PRIMO).

En el programa 8.17 se observa un ejemplo de uso de la función primo. El programa principal evalúa si la variable a es prima y, de ser cierta esta situación, envía al Puerto D del microcontrolador el valor 0xF0, en caso contrario, envía el valor 0x05. Como se puede observar en la instrucción *b = primo(a)*; están involucrados todos las instrucciones de la estructura de la función. En este ejemplo, *b* = 0, ya que 34 no es un número primo.

Programa 8.17. Declaración y uso de la función detectora de números primos

```
#define ES_PRIMO 1
#define NO_ES_PRIMO 0
unsigned char primo(unsigned char num);
unsigned char a,b;
void main()
{
        TRISD = 0;
        a = 34;
        b = primo(a);
        if(b == ES_PRIMO) LATD = 0xF0;
        else              LATD = 0x05;
        while(1);
}
unsigned char primo(unsigned char num)
{
        unsigned char i = 0;
        if(num<=3) return(ES_PRIMO);
        for(i=2;i<num;i++)
        {
                if(num%i==0) return(NO_ES_PRIMO);
        }
        return(ES_PRIMO);
}
```

A continuación se muestra otro ejemplo de una función que determina si un número es o no es capicúa. Un número capicúa es aquel que, a pesar de que sus cifras se invierten de posición, sigue representando la misma magnitud. Por ejemplo, el número 121 es capicúa, ya que, si se invierten de posición sus cifras, sigue representando el valor 121. El prototipo de la función y la estructura de la función se muestran en el programa 8.18. Básicamente, la función capicúa recibe un número de 8 bits (tres dígitos en base decimal) y analiza si la cifra de las centenas es igual al de las unidades, con lo cual el número resultaría capicúa.

Programa 8.18. Ejemplo de una función de detección de un número capicúa

```
#define ES_CAPICUA 2
#define NO_ES_CAPICUA 3
unsigned char capicua(unsigned char num);
unsigned char a,b;
void main()
{
        TRISD = 0;
        b = capicua(131);
        if(b == ES_CAPICUA) LATD = 0xFF;
        else                LATD = 0xA0;
        while(1);
}
unsigned char capicua(unsigned char num)
{
        unsigned char C,U;
        C = num/100;
        U = (num%100)%10;
        if(C==U) return(ES_CAPICUA);
        else return(NO_ES_CAPICUA);
}
```

Dentro del ámbito de las funciones se puede utilizar un modificador para los tipos de variables locales, conocido como static. Por defecto, todas las variables locales creadas en la función pierden su valor al salir de esta. Por ejemplo, en la función capicúa, las variables *C* y *U* se crean cada vez que se invoca la función. En el programa 8.18 la variable *C* toma el valor de 1 y *U* también, pero una vez que la función culmina, estas variables pierden sus valores. Si se volviera a invocar la función capicúa, *C* y *U* se volverían a declarar teniendo como valores iniciales 0x00. En algunas situaciones se necesita que las variables locales mantengan el valor que tenían antes de salir del último llamado de la función. Esto implicaría utilizar variables globales dentro de la función. Otra opción sería hacer uso del modificador static de la siguiente manera:

```
        static unsigned char C,U;
```

Entonces, si se volviera a hacer uso de la función capicúa, los valores de *C* y *U* desde un inicio serían *C* = 1 y *U* = 1. La variable estática se comporta como una variable global, pero sigue siendo local, ya que solo puede ser usada por la función. Algunas funciones requieren el uso de este tipo de variables, como se verá más adelante en algunos ejemplos de aplicación.

Si ahora se desea trabajar un proyecto que utilice tanto la función capicúa como la función primo, hacer uso de ambas funciones en el programa principal donde se encuentra la función *main()* resul-

taría en un código muy extenso. Por tal motivo, como ya se tienen elaboradas las funciones, estas se pueden agrupar en una librería, a la que se le puede dar el nombre de *Cifras*.

Toda librería en ANSI C se compone de dos archivos: uno de extensión .h y otro de extensión .c. El primer archivo, conocido como «cabecera», contiene directivas y redefiniciones de tipos de datos junto con los prototipos de las funciones que componen la librería. El segundo archivo contiene la declaración de variables y las estructuras de las funciones cuyos prototipos se encuentran en el archivo de extensión .h. Ambos archivos deben ser guardados con el mismo nombre y solo deben variar la extensión.

Para crear la librería primero se deberá generar un nuevo archivo de cabecera *(Header File)*. Para esto, se selecciona la opción *File->New File->C->Header File* en MPLAB X. En este archivo se deben incluir las definiciones de alias con la directiva #define y también todos los prototipos de las funciones. Una vez hecho esto, se debe guardar el archivo de texto con el nombre Cifras.h, ya que la librería se ha denominado Cifras. El archivo debe ser almacenado en la carpeta del proyecto en MPLAB X creado. En el programa 8.19 se muestra el contenido de este archivo.

Programa 8.19. Contenido del archivo Cifras.h correspondiente a la librería Cifras

```
#define ES_PRIMO          1
#define NO_ES_PRIMO       0
#define ES_CAPICUA        2
#define NO_ES_CAPICUA     3

unsigned char capicua(unsigned char num);
unsigned char primo(unsigned char num);
```

Posteriormente, se deberá crear el archivo Cifras.c, el cual contiene la estructura de las funciones. Para esto, se selecciona la opción *File->New File->C->C Source File* en MPLAB X. Es importante incluir en la cabecera de este archivo la directiva #include “Cifras.h”, ya que se requiere de los prototipos de las funciones y de la declaración de los alias para que estas funciones tengan sentido, pues hacen uso de los alias creados. En el programa 8.20 se muestra el contenido del archivo Cifras.c.

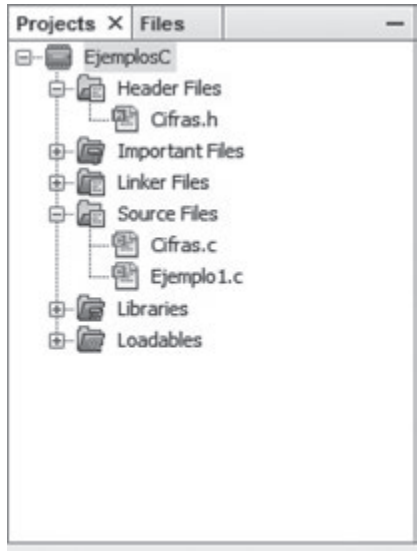
Programa 8.20. Contenido del archivo Cifras.c correspondiente a la librería Cifras

```
#include "Cifras.h"
unsigned char primo(unsigned char num)
{
        unsigned char i = 0;
        if(num<=3) return(ES_PRIMO);
        for(i=2;i<num;i++)
        {
```

```
                if(num%i==0) return(NO_ES_PRIMO);
        }
        return(ES_PRIMO);
}
unsigned char capicua(unsigned char num)
{
        unsigned char C,U;
        C = num/100;
        U = (num%100)%10;
        if(C==U) return(ES_CAPICUA);
        else return(NO_ES_CAPICUA);
}
```

Una vez creados ambos archivos en las carpetas del proyecto, la librería queda lista, tal como se muestra en el gráfico 8.9.

Gráfico 8.9. Panel de archivos con la librería Cifras agregada al proyecto



Elaboración propia.

Es así que el programa principal que se encuentra en el archivo Ejemplo1.c ya puede hacer uso de las funciones de la librería Cifras. En el programa 8.21 se muestra el contenido de este archivo. Es importante incluir en la cabecera los archivos de las librerías para que estas sean reconocidas por el programa principal. El nombre de la librería de extensión .h es el que debe ser incluido en la cabecera, y debe ir entre comillas dobles, lo cual implica que el archivo se encuentra en la carpeta del proyecto. Se recomienda que las librerías creadas por el usuario se incluyan luego de la definición de las directivas #pragma de configuración de las opciones de configuración inicial del microcontrolador.

Programa 8.21. Archivo principal Ejemplo1.c que hace uso de la librería Cifras

```
#include <xc.h>

#pragma config FOSC          = XTPLL_XT   //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV        = 1          //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV        = OSC1_PLL2  //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT          = ON
#pragma config WDT           = OFF
#pragma config LVP           = OFF
#pragma config BOR           = OFF
#pragma config DEBUG         = ON
#pragma config VREGEN        = OFF
#pragma config PBADEN        = OFF
#pragma config MCLRE         = ON
#include "Cifras.h"

unsigned char a,b;
void main()
{
        TRISD = 0;
        a = 34;
        b = primo(a);
        if(b == ES_PRIMO) LATD = 0xF0;
        else              LATD = 0x05;
        b = capicua(131);
        if(b == ES_CAPICUA) LATD = 0xFF;
        else          LATD = 0xA0;
        while(1);
}
```

Como se puede observar, el programa principal invoca a las funciones primo() y capicua(), las cuales se encuentran declaradas y estructuradas en la librería. Esto simplifica enormemente la programación, ya que evita el exceso de código en el archivo principal Ejemplo1.c y permite que este se muestre más ordenado. Las librerías se pueden compartir entre diversos programadores, lo cual facilita el trabajo en equipo y la compatibilidad de código al momento de hacer la integración de múltiples programas que corresponden al mismo proyecto.

8.7 Arreglos y punteros

Los arreglos son uno de los pilares del lenguaje ANSI C, ya que permiten la manipulación de múltiples valores numéricos asociados a una sola variable, denominada «vector». Los arreglos y punteros hacen uso intensivo de la memoria de datos, pues normalmente se requiere el almacenamiento en más de una dirección de memoria.

Es así que los arreglos interactúan con el dato y su dirección. El uso de arreglos es vital para la implementación de algoritmos avanzados, tales como los usados en procesamiento digital de señales. Además, facilitan la creación de mensajes y tablas de datos que pueden ser muy fácilmente utilizadas en el programa.

Un arreglo de datos se declara como una variable, indicando entre corchetes ([ ]) la cantidad de elementos que la componen. Por ejemplo, para definir un arreglo se pueden utilizar los siguientes formatos:

```
unsigned char vector[10];
char mensaje[] = “Programando el PIC18F4550”;
unsigned int valores[] = {0x9040,0x1234,0x6798,0x1A2B,0xF761};
```

En el primer caso, el arreglo vector se declara de tipo unsigned char (8 bits sin signo) y de un tamaño de 10 elementos. Esto implica que el arreglo vector ocupará 10 espacios de memoria, donde cada espacio será de 1 byte de capacidad. En total serían 10 bytes de memoria reservados para el arreglo vector. En el segundo caso, el vector mensaje no tiene indicado el número de elementos, pero en su inicialización se le asigna la cadena de caracteres ASCII “Programando el PIC18F4550”, la cual se encuentra comprendida por 25 bytes, lo que implica que mensaje es un vector de 25 posiciones de memoria más un carácter nulo o de término que indica el fin de la cadena (en total se reservarían 26 espacios de memoria). Finalmente, el arreglo valores está compuesto de 5 elementos donde cada elemento es de 2 bytes y ocupa dos direcciones de memoria. Cuando un arreglo se inicializa con valores no ASCII, todos los elementos se deberán colocar entre llaves y separados por una coma.

Los arreglos pueden declararse tanto en la memoria de datos como en la memoria de programa. Según la siguiente declaración, se crean dos vectores, uno de cinco elementos y 8 bits por elemento y otro de cuatro elementos y 16 bits por elemento. En el gráfico 8.10 se muestran cómo los datos de inicialización estarían almacenados en la memoria de datos o SRAM. El formato para los datos de 16 bits es “*endianness*”, donde el byte de más peso ocupa la dirección más alta.

```
char mensaje[] @0x100;
int  vector[]  @0x106;
void main()
{
   mensaje[0]='P'; mensaje[1]='I'; mensaje[2]='C'; mensaje[3]='1';
   mensaje[4]='8'; mensaje[5]=0x00;
   vector[0]=0xA091; vector[1]=0x1234; vector[2]=0xB751;
   vector[3]=0x45F1;
   while(1);
}
```

Cuando se declara una variable en memoria SRAM y se utiliza el modificador @ para especificar la dirección, no es posible inicializar dicha variable en la declaración. Por tal motivo, en el código mostrado la inicialización de la variable se tiene que realizar dentro de la función *main()*.

Gráfico 8.10. Alojamiento de los datos de los arreglos mensaje y vector en la memoria de datos

0x100	'P'
0x101	'I'
0x102	'C'
0x103	'1'
0x104	'8'
0x105	0x00
0x106	0x91
0x107	0xA0
0x108	0x34
0x109	0x12
0x10A	0x51
0x10B	0xB7
0x10C	0xF1
0x10D	0x45

Elaboración propia.

De la misma manera, si un arreglo se quiere declarar en la memoria de programa (por lo cual ninguno de sus elementos podría ser modificado), se puede hacer uso de la siguiente declaración:

```
const char TABLA[] @0x5000 = "Hola Mundo";
```

Solamente en el caso exclusivo de la declaración de arreglos de tipo const (en memoria FLASH), se puede realizar la inicialización mientras se asigna con el modificador @ la dirección que tomará el arreglo en la memoria FLASH. Esto es una característica específica del compilador XC8.

El arreglo tabla se compone de 10 bytes, y estos se encuentran almacenados en la memoria de programa entre las direcciones 0x5000 y 0x5009. Cada carácter de tabla puede ser accedido en el programa principal, pero solo para el proceso de lectura, mas no de escritura. El compilador arrojaría un error si se tratara de realizar el último caso.

Otra variante de los arreglos es la declaración bidimensional o en forma matricial. Aquí el arreglo presenta dos índices: el número de filas y el número de columnas. Para declarar un arreglo de forma matricial se sigue el siguiente formato:

```
unsigned char matriz[3][4];
```

La variable matriz presentará tres filas y 4 columnas, donde cada elemento de esta matriz es de un byte sin signo. En total este arreglo ocupa 12 bytes de la memoria de programa. No es recomendable el uso intensivo de este tipo de arreglo, ya que el consumo de memoria se multiplicaría.

El acceso a la información de los arreglos se realiza mediante su índice. El índice representa la posición de cada elemento del arreglo y siempre debe estar representado por una cifra entera positiva. El primer valor del índice es el 0 y el último es el tamaño del arreglo menos uno. Por ejemplo, en el programa 8.22 se muestra un ejemplo de acceso a un arreglo declarado en la memoria de datos.

Programa 8.22. Ejemplo de acceso a la información contenida en un arreglo de datos

```
unsigned char vector[]="Hola Mundo";
unsigned char a,b;
void main()
{
        a = vector[0];
        b = 'k';
        vector[5] = b;
        while(1);
}
```

En el ejemplo mostrado, la variable *a* toma el valor ASCII 'H', que es el primer elemento del arreglo vector (índice = 0). Luego, la variable *b* toma el valor ASCII 'k' y este valor se asigna al sexto elemento del arreglo (índice = 5), el cual reemplaza el valor 'M' por 'k'.

Por otro lado, los punteros son más versátiles que los arreglos, ya que permiten reasignar las direcciones en las cuales se van a almacenar sus datos. La utilidad de los punteros reside en la capacidad para apuntar a determinada dirección de memoria y extraer o guardar un dato en dicha posición. Con los punteros se pueden reutilizar las direcciones de memoria que han sido asignadas previamente a otras variables.

La declaración de un puntero sigue este formato:

```
Tipo *nombre del puntero;
```

El tipo implica el tamaño del registro de memoria al cual va a apuntar el puntero. Por ejemplo, puede apuntar a registros de tamaño de 8 bits, 16 bits, 24 bits o 32 bits. El nombre del puntero debe ir precedido por un * durante su declaración. El acceso a un puntero también se puede realizar mediante índices (al igual que un arreglo) o mediante el acceso al contenido, precediendo * al nombre del puntero. Por defecto, un puntero apunta a un dato de 16 bits.

Para asignar la dirección a un puntero se debe hacer uso del operador &. Por ejemplo, si una variable *a* es declarada de tipo unsigned char e inicializada con el valor de 9, y la variable *a* ocupa la dirección 0x60 de la memoria SRAM. Dicha dirección contiene el valor 9, ya que se está asignando esta cifra a la variable en su declaración. Si posteriormente se declara un puntero de nombre *p* de la siguiente manera y se hace la siguiente asignación:

```
unsigned char a=9;
unsigned char *p;
void main()
{
        p = &a;
        *p=10;
}
```

En este ejemplo, *p* es un puntero que apunta a registros de 8 bits (porque está declarado de tipo unsigned char). El puntero tiene asignada la dirección de la variable a (p = &a), por lo cual apuntará a la dirección 0x60. Posteriormente, se carga el valor de 10 al contenido apuntado por *p* (instrucción **p = 10*), el cual reemplaza al valor de 9 que tenía la variable *a*. En otras palabras se ha cambiado el contenido de la variable a usando el puntero *p*.

Por ejemplo, en el programa 8.23 se muestra la declaración de un puntero y cómo este interactúa con el arreglo de nombre vector previamente declarado.

Programa 8.23. Ejemplo de manejo de punteros

```
unsigned char vector[]="Hola Mundo";
unsigned char *p;
void main()
{

        p  = &vector[5];
        *p = 'R';
        p++;
        *p = 'e';
        p[2]='g';
        while(1);
}
```

En el programa se crea un arreglo de nombre vector de 10 elementos inicializado con el mensaje "Hola Mundo". El compilador XC8 por defecto asignará la dirección 0x01 de la SRAM al primer elemento de vector. Posteriormente se crea el puntero *p* que apunta a valores de 8 bits. Cuando se asigna la dirección del sexto elemento de vector a *p*, este apunta a la dirección 0x05. Inmediatamente, al contenido de esta dirección tendrá asignado el código ASCII 'R', reemplazando al ASCII 'M' que estaba en esa posición. Luego, el puntero se auto incrementa en 1 dígito, apuntando a la dirección 0x06. Cuando se asigna el valor ASCII al contenido apuntado en *p* (instrucción **p = 'e'*), este reemplaza al valor ASCII 'u' que se encontraba en esa posición. Finalmente, el tercer elemento de *p* es asignado con el valor 'g', lo cual implica que la dirección 0x06+0x02 = 0x08 toma el valor ASCII 'j' reemplazando al valor ASCII 'd'. De esta manera, el mensaje inicial cargado en vector cambia a ser "Hola Rengo".

8.8 Estructuras y uniones

Las estructuras son arreglos de datos que se pueden invocar desde una sola variable al igual que los punteros. La diferencia principal con estos últimos radica en que los elementos dentro del arreglo pueden ser de diferente tipo o tamaño. Por ejemplo, se desea crear una variable de nombre *Persona*, la cual tiene asociados los siguientes datos: edad, peso, talla, DNI, nombre. Cada dato de la variable *Persona* presenta diferentes características:

1. La edad de la persona es un valor de 8 bits, ya que puede oscilar entre 0 y 255 años.
2. El peso de la persona también es un valor de 16 bits y puede alternar entre 0 y una cifra mayor a 255 kg.
3. La talla de la persona también es un valor de 8 bits y puede alternar entre 0 y 255 centímetros.
4. El DNI de la persona es un valor de 32 bits sin signo, ya que se consideran 7 cifras numéricas.
5. El nombre de la persona es una cadena de caracteres de N dígitos.

Queda claro que se tienen cuatro tipos diferentes de variable. El primero es de tipo unsigned char, el segundo unsigned int, el tercero unsigned char, el cuarto unsigned long y el último un puntero a una cadena de caracteres tipo char. ¿Cómo crear una variable que englobe estos cuatro tipos de datos distintos?

Una estructura es la solución a este problema. Basta con declararla de la siguiente manera:

```
struct{
                unsigned char edad;
                unsigned int    peso;
                unsigned char talla;
                unsigned long DNI;
                char *nombre;
        }Persona;
```

La variable *Persona* es una estructura y contiene cinco campos. Dos de ellos ocupan 1 byte cada uno (edad y talla), mientras que el segundo ocupa 2 bytes, el cuarto ocupa 4 bytes y el último campo, al ser un puntero a 8 bits, ocupa solo dos bytes. En total, la estructura *Persona* requiere de 10 bytes de la memoria SRAM.

Para hacer uso de la variable *Persona* solo basta declarar la estructura e invocar cada uno de sus campos. En una estructura el campo se especifica colocando el nombre de la variable seguido por un punto y luego el nombre del campo. Por ejemplo, para actualizar la altura de la variable *Persona* se ejecuta la instrucción: *Persona.talla = 170*. En el programa 8.24 se muestra un ejemplo de creación de una estructura y la actualización de sus campos.

Programa 8.24. Ejemplo de declaración y actualización de datos de una estructura

```
struct{
                unsigned char edad;
                unsigned char peso;
                unsigned char talla;
                unsigned long DNI;
                char *nombre;
        }Persona;
char name[] = "Sergio Salas A.";
void main()
{
        Persona.edad= 31;
        Persona.peso= 81;
        Persona.talla= 180;
        Persona.DNI = 40765319;
        Persona.nombre = &name[0];
        while(1);
}
```

También es posible inicializar una estructura durante su inicialización de la misma manera que se ha hecho en el caso de una variable o un arreglo matricial. Para esto, los campos iniciales se colocan entre { }, tal como se muestra en el siguiente ejemplo:

```
struct{
        unsigned char edad;
        unsigned char peso;
        unsigned char talla;
        unsigned long DNI;
        char nombre[20];
}Persona={31,81,180,40765319,"Sergio Salas A."};
```

Una de las ventajas que ofrece la estructura es la posibilidad de declarar campos de longitudes variables. Es decir, se pueden crear campos de un bit, dos bits, tres bits o cualquier tamaño a definir hasta un máximo de 8 bits. El formato para determinar el tamaño del campo es colocar el nombre de este seguido de dos puntos (:) y el valor en bits. Cabe mencionar que este procedimiento solo es válido en el ámbito de una estructura. Por ejemplo, se tiene la siguiente declaración de estructura:

```
struct{
                char bit0: 1;
                char bit1: 1;
                char bit2: 1;
                char bit3: 1;
                char bit4: 1;
                char bit5: 1;
                char bit6: 1;
                char bit7: 1;
        }_bits;
```

La variable *_bits* posee 8 campos internos, cada campo tiene un bit de tamaño. Esto es interesante, ya que la estructura _bits solo ocupa 1 byte y presenta 8 campos: uno para cada bit de dicho byte. El bit menos significativo (bit0) se declara primero, mientras que el bit más significativo (bit7) se declara al final. Si esta estructura se introduce en la memoria de datos del PIC18, entonces solo ocuparía una única dirección. En el programa 8.25 se muestra un ejemplo de manipulación de una estructura.

Programa 8.25. Ejemplo de manejo de estructuras con campos de 1 bit de longitud

```
struct{
        char bit0: 1;
        char bit1: 1;
        char bit2: 1;
        char bit3: 1;
        char bit4: 1;
        char bit5: 1;
        char bit6: 1;
        char bit7: 1;
        }_bits;
char car;
char *p;
void main()
{
  _bits.bit0 = 1;
  _bits.bit1 = 1;
  _bits.bit2 = 0;
  _bits.bit3 = 0;
  _bits.bit4 = 0;
  _bits.bit5 = 0;
  _bits.bit6 = 0;
  _bits.bit7 = 1;
  p=&_bits;
  car = *p;
  while(1);
}
```

En el programa 8.25 se muestra una estructura de nombre _bits cuyos 8 campos son de 1 bit de longitud. Esta estructura se encuentra declarada en la dirección 0x04 por defecto de la memoria de datos. La siguiente variable declarada car ocupa la dirección siguiente 0x03. Posteriormente, se declara el puntero *p en la dirección por defecto.

Cuando el programa inicia, se actualizan los 8 campos binarios de la estructura _bits con los siguientes valores 10000011_2. Esto significa que la dirección 0x04 de la memoria de datos automáticamente (después de la actualización del campo _bits.bit7) contiene el valor 0x83. Luego, el puntero *p* toma la dirección de la estructura _bits, que es 0x04. Luego, la variable car toma el contenido apuntado por *p*, con lo cual *car* toma el valor 0x83.

Ahora se analizará el programa 8.26, que es una copia fiel del programa 8.25. Se ha incluido un pequeño cambio, que corresponde a un campo más a la estructura: bit8y9: 2. ¿Qué cambios ocurrirán en el programa?

Programa 8.26. Ejemplo de manejo de estructuras con más de ocho campos de 1 bit

```
struct{
        char bit0: 1;
        char bit1: 1;
        char bit2: 1;
        char bit3: 1;
        char bit4: 1;
        char bit5: 1;
        char bit6: 1;
        char bit7: 1;
        char bit8y9: 2;
        }_bits @0x60;
char car;
char *p;
void main()
{
        _bits.bit0 = 1;
        _bits.bit1 = 1;
        _bits.bit2 = 0;
        _bits.bit3 = 0;
        _bits.bit4 = 0;
        _bits.bit5 = 0;
        _bits.bit6 = 0;
        _bits.bit7 = 1;
        _bits.bit8y9 = 2;
        p=&_bits;
        p++;
        *p|=0x40;
        car = *p;
        while(1);
}
```

Ahora los resultados son diferentes. En primer lugar, la estructura _bits posee un campo adicional: bit8y9 que es de 2 bits, con lo cual toda la estructura ocupa 10 bits de longitud. Debido a esto, la estructura deberá ocupar dos direcciones de la memoria de datos. Los primeros 8 bits menos significativos ocuparán la dirección 0x60, mientras que el campo _bits.bit8y9 ocupará los dos bits menos significativos de la dirección 0x61.

¿Qué ocurre con los 6 bits más significativos de la dirección 0x611? Pues, mediante la estructura, no es posible acceder a ellos. Tendría que utilizarse un puntero para poder apuntar a dicha dirección y modificar el valor de 8 bits de esa posición.

En el programa principal se vuelve a cargar la dirección 0x60 con el valor 0x83 y el campo _bits. bit8y9 se carga con 2 (10_2), lo cual modifica el contenido de los dos bits menos significativos de la posición 0x61. Los 6 bits más significativos de esta dirección permanecen en 000000_2, con lo cual el valor obtenido es 0x02. Luego, el puntero toma la dirección 0x60 y se incrementa en un dígito, apuntando a la dirección 0x61, a cuyo contenido realiza la operación OR con 0x40 generándose el resultado 0x40 OR 0x02 = 0x42. Finalmente, la variable car toma el valor de la dirección 0x61, con lo cual adquiere el valor 0x42.

Por otro lado, ¿qué pasaría si se quisiera tener acceso a cada bit de un registro y al mismo tiempo poder acceder a todo el valor de dicho registro? En los ejemplos mostrados en los programas 8.26 y 8.25 se vio que para tal efecto se tenía que reasignar la dirección de la estructura a un puntero, pero esto requiere de carga computacional adicional.

Las uniones son otro tipo de dato del ANSI C que contribuyen a solucionar el problema presentado. Se tiene el siguiente programa en lenguaje ensamblador:

```
movlw 0x88
movwf LATD
bsf LATD,0
```

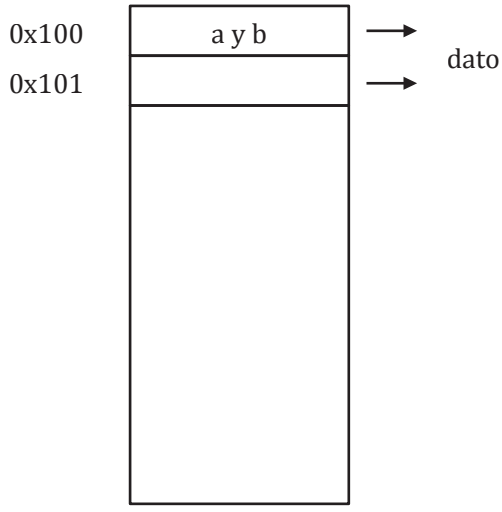
Se puede observar de esta rutina en ensamblador que el registro LATD ha sido accedido de dos formas diferentes. Primero, se ha cargado el valor 0x88 en todo su registro de 8 bits de longitud. Luego, solo se ha afectado el bit LSB, cargándose el nivel '1' y modificando la cifra completa del registro de 0x88 a 0x89. Este mismo efecto se puede lograr haciendo uso de una unión.

Las uniones en ANSI C presentan una sintaxis muy parecida a la estructura. Obsérvese la siguiente unión:

```
union{
        unsigned char a;
        unsigned char b;
        int     dato;
        }campos @0x100;
```

La variable campos es una unión de tres elementos: uno de 8 bits de nombre *a*, otro de 8 bits de nombre *b* y un campo de 16 bits de nombre *dato*. En una unión, los campos declarados ocupan el mismo espacio en memoria. Es decir, *a* y *b* al ser de la misma longitud toman posición de la dirección 0x100 de la memoria de datos. En cambio el campo *dato*, al ser de 16 bits, debe ocupar dos direcciones. Su parte baja ocupa la dirección 0x100, mientras que sus 8 bits más significativos ocuparán la posición 0x101, tal como se observa en el esquema del gráfico 8.11.

Gráfico 8.11. Esquema de asignación de direcciones a los campos de una unión en la memoria de datos



Elaboración propia.

De tal manera, acceder a la variable *a* o *b* implica reutilizar la misma dirección de memoria. De igual forma, la parte baja del campo dato apunta a la misma dirección. Esto ofrece una gran ventaja, ya que, como se verá más adelante, se podrá programar un código equivalente al visto en la rutina de ensamblador de la página anterior. En el programa 8.27 se crean los tres campos: *a*, *b* y *dato*. En la función principal *main()* se asignan tres valores diferentes a cada campo. Al final, cada asignación sobreescribe el valor de la variable anterior. Por ejemplo, la asignación de *campos.b* reemplaza al valor del *campos.a* y, finalmente, el byte menos significativo de *campos.dato* reemplaza a las dos variables anteriores con el valor 0x34.

Programa 8.27. Ejemplo de manipulación de campos de una unión

```
union{
        unsigned char a;
        unsigned char b;
        int dato;
        }campos @0x100;
unsigned char car;
void main()
{
        campos.a = 0x50;
        campos.b = 0x12;
        campos.dato = 0x7434;
        car = campos.a;
        while(1);
}
```

Es así que, al finalizar el programa, la variable campos.a = 0x34, campos.b = 0x34, campos.dato = 0x7434 y car = 0x34.

Lo interesante resulta al combinar la estructura y la unión. Regresando al ejemplo en lenguaje ensamblador, se quiere crear una variable de nombre PUERTOB cuyos campos son _Byte de tipo char y RB0, RB1, RB2, RB3, RB4, RB5, RB6 y RB7, cada uno de un 1 bit. Cuando uno de los campos de un bit se actualiza, inmediatamente esto se ve reflejado en el campo _Byte. Eso es lo mismo que ocurre en el ejemplo mostrado en ensamblador. En el programa 8.28 se muestra este ejemplo.

Programa 8.28. Ejemplo de combinación de una estructura y una unión

```
union{
  struct{
        char RB0:1;
        char RB1:1;
        char RB2:1;
        char RB3:1;
        char RB4:1;
        char RB5:1;
        char RB6:1;
        char RB7:1;
  }_bits;
  char _Byte;
}PUERTOB @0x100;
unsigned char car;
void main()
{
        PUERTOB._Byte = 0xA5;
        PUERTOB._bits.RB1 = 1;
        PUERTOB._bits.RB7 = 0;
        car = PUERTOB._Byte;
        while(1);
}
```

En el programa se observa que la variable PUERTOB es declarada como una unión. Esta presenta dos campos internos: _Byte y la estructura _bits. Al estar ambos campos dentro de la unión y al tener la misma dimensión (la estructura presenta 8 campos de 1 bit cada uno, es decir, 8 bits en total), ocupan la misma dirección en la memoria de datos (definida en el valor 0x100).

En el programa principal, se asigna el valor 0xA5 al campo _Byte de la variable PUERTOB, con lo cual la dirección 0x100 se carga con este valor. Luego, el campo RB1 de la estructura _bits se pone a nivel '1', con lo cual el valor de la dirección 0x100 cambia a 0xA7. Luego, el campo RB7 de la estructura se pone a '0', quedando como valor final el valor 0x27, el cual se carga en la variable *car*.

El uso de esta combinación de estructuras y uniones es muy útil para diversas aplicaciones en las cuales se quieren manipular los bits de un registro de modo independiente. Esto es muy útil para aplicaciones de conversión serial a paralelo, como se verá más adelante en la aplicación de interfaz con un teclado PS/2, o también en el siguiente capítulo, cuando se vea la detección de teclas pulsadas de un control remoto infrarrojo para televisión.

8.9 Tipos de variables compuestos

Para mejorar la legibilidad de un programa, muchas veces es necesario aclarar los tipos de variable a usar. Por ejemplo, el tipo *int* se sabe representa una variable de 16 bits. ¿Queda claro esto para un programador que hace uso de otro compilador para programar otra marca de sistema embebido? La respuesta es no. Lo más probable es que el programador dude si el tipo *int* representa 8 o 16 bits. Para dejar en claro qué se está declarando se puede hacer uso del operador *typedef*.

Al redefinir un tipo de variable, se puede dejar muy en claro qué se quiere representar. Por ejemplo, si se declara una variable *num* de tipo *int* y se quiere dejar muy presente que esta variable es de 16 bits con signo, se puede hacer lo siguiente:

```
typedef signed int_16bits_con_signo;
_16bits_con_signo num;
```

Entonces, queda claro que la variable *num* corresponde a un tipo de tamaño de 16 bits con signo. El programador sabrá, de acuerdo con su experiencia, cuál es el tipo de dato que corresponde a estas características, y, de esta manera, tendrá una comprensión más clara del programa.

Del mismo modo, las uniones y estructuras se pueden convertir a nuevos tipos compuestos, de forma que una estructura muy necesitada se puede definir a través del operador *typedef* y, así, crear nuevas variables.

Como ejemplo, se tienen conectados a cada uno de los pines RB0, RB1, RB2 y RB3 un sensor magnético que opera como alarma de seguridad para cuatro ventanas. Cuando una de las ventanas se abre, el sensor envía un nivel ‘1’ al pin respectivo. Dependiendo de la ventana (ventana 1, 2, 3 o 4), se requiere generar un código de alarma de 4 bits que será transmitido a un dispositivo externo que recibe este valor binario a través de los pines RB4, RB5, RB6 y RB7 del PIC18F4550. Si la alarma es detectada por el pin RB0, el código a enviar es 0x08; si se detecta por RB1, el código será 0x0E; si se detecta en el pin RB2, el código es 0x0A; y por el pin RB3, el código será 0x02. En el programa 8.29 se muestra un ejemplo, en el que se crea un nuevo tipo de variable compuesto por una estructura dentro de una unión. Los campos de la estructura presentan un nombre asociado a su función, lo cual hace más entendible el programa.

Programa 8.29. Ejemplo de un tipo de dato compuesto para una aplicación de detección de alarmas

```
typedef union
        {
        struct{
                char alarma1:1;
                char alarma2:1;
                char alarma3:1;
                char alarma4:1;
                char cod_alarma:4;
        }info_alarma;
        char estado;
}alarmas;
alarmas puertob;
void main()
{
        TRISB = 0x0F;
        while(1)
        {
                puertob.estado = PORTB;
                if(puertob.info_alarma.alarma1 = 1)
                        puertob.info_alarma.cod_alarma = 8;
                elseif(puertob.info_alarma.alarma2 = 1)
                        puertob.info_alarma.cod_alarma = 14;
                elseif(puertob.info_alarma.alarma3 = 1)
                        puertob.info_alarma.cod_alarma = 10;
                elseif(puertob.info_alarma.alarma4 = 1)
                        puertob.info_alarma.cod_alarma = 2;
                else
                        puertob.info_alarma.cod_alarma = 0;
                LATB = puertob.estado;
        }
}
```

8.10 Uso del lenguaje ensamblador dentro de ANSI C

Muchas veces, al programar un sistema embebido, se desea que el programa ejecute cierta rutina en un tiempo bastante corto. Cuando esta rutina se diseña en ANSI C, no se sabe realmente si se cumplirán los requerimientos de tiempo. Es posible que el compilador genere un código que no cumpla con las expectativas. Sin embargo, el programador conoce la solución en lenguaje ensamblador, pero no desea desarrollar todo el proyecto en este lenguaje. Para solucionar este problema, el compilador MPLAB XC8 permite la inclusión de código en ensamblador dentro del programa en ANSI C. No obstante, existen algunas restricciones y diferencias con la programación vista en los primeros capítulos del libro.

Para incluir un código en lenguaje ensamblador se deben utilizar dos directivas: #asm y #endasm. La primera indica al compilador que el código que viene a continuación es ensamblador, y la segunda directiva implica el fin de la rutina y, por tanto, lo que viene después debe estar escrito en ANSI C. En el siguiente código se observa un ejemplo:

```
#asm
        movlw 0x10
        movwf 0x30,0
ESPERA:
        decfsz 0x30,1,0
        goto  ESPERA
#endasm
```

Por otro lado, el compilador XC8 permite realizar el uso de instrucciones en ensamblador mediante la función *asm( )*. Es necesario que se coloque el punto y coma al final, ya que esta operación es una función de ANSI C. Esta función permite introducir como argumento la instrucción en ensamblador como si fuera una cadena de texto, de la siguiente manera:

```
asm("movlw 0x40");
```

Como se puede observar, las instrucciones en ensamblador mantienen su formato normal. Sin embargo, se deben tener en cuenta las siguientes restricciones:

1. Todas las etiquetas deben terminar en dos puntos (:).
2. Los comentarios mantienen el formato del ANSI C. Es decir, haciendo uso del operador // o /*. No debe colocarse punto y coma después de una instrucción.
3. No se permite utilizar directivas en ensamblador.
4. Las instrucciones se deben redactar con la sintaxis completa.
5. No es posible utilizar las variables creadas en ANSI C en ensamblador, pero sí es posible utilizar los nombres de los registros de función especial.
6. Para las instrucciones de manejo de tablas, se debe usar el mnemónico completo. Estos son los mnemónicos válidos:

- TBLRD
- TBLRDPOSTDEC
- TBLRSPOSTINC
- TBLRDPREINC
- TBLWT
- TBLWTPOSTDEC
- TBLWTPOSTINC
- TBLWTPREINC

7. El formato de las bases numéricas por defecto es decimal igual que en ANSI C.

En el programa 8.30 se muestra un ejemplo de una rutina en ensamblador que interactúa con las tablas y variables creadas en ANSI C. Este programa hace uso de los modos de direccionamiento directo para manipular las variables, y de las instrucciones de manejo de tabla. Cada uno de los caracteres de la tabla es enviado al registro LATD.

Programa 8.30. Ejemplo de inclusión de rutinas en ensamblador dentro de ANSI C

```
const char NOMBRE[] @0x7000 = "Usando ensamblador en C\n";
unsigned char i @0x01;
unsigned char p @0x02;
unsigned char a @0x200;
void main() @0x40
{
        TRISD = 0;
        a = 0x30;
        #asm
                movlw 0x65
                movwf 0x01,0            //i = 0x65
                movff 0x01,0x02         //p = i
                clrf  TBLPTRU           //TBLPTRU = 0x00
                movlw 0x70
                movwf TBLPTRH,0         //TBLPTRH = 0x70
                clrf  TBLPTRL           //TBLPTRL = 0x00
  LECTURA:
                TBLRDPOSTINC            //Lectura de tabla y auto incremento de TBLPTR.
                movf  TABLAT,0,0        //W = TABLAT.
                movwf 0x02,0            //p = W.
                movlw 0x0A              //W = '\n'.
                subwf 0x02,0,0          //Comparando p con el carácter de término ' \n'.
                btfsc STATUS,2,0        //El bit 2 de STATUS es el Z.
                bra   SALIR             //Si p == '\n' se termina la rutina en ensamblador.
                movf  0x02,0,0          //W = p
```

```
        movwf LATD,0            //LATD = W
        bra LECTURA
SALIR:
    #endasm
        a = NOMBRE[5];          //a = 'o'
        while(1);
}
```

8.11 Manejo de interrupciones en MPLAB XC8

En el capítulo 6 se mostró que el PIC18F contiene dos vectores de interrupción: uno de alta prioridad y otro de baja prioridad. El primero se halla en la dirección 0x08 de la memoria de programa, mientras que el segundo lo hace en la dirección 0x018. En cada vector se ubicaba una instrucción de salto a una rutina de interrupción; luego se ejecutaban las instrucciones de cada rutina y, finalmente, se retornaba al programa principal con la instrucción retfie.

En el leguaje ANSI C el esquema de funcionamiento de las interrupciones es similar. Se tienen dos vectores de interrupción y en cada uno de ellos se debe invocar a una nueva rutina de interrupción. La diferencia radica en que las rutinas de interrupción ahora serán funciones, y la invocación de estas se realizará haciendo uso del lenguaje ensamblador.

Las funciones de interrupción, como tales, requieren de un prototipo y definición de estructura. Esto tiene que estar especificado en el programa. La dirección del vector de interrupción se define al momento de crear la función de interrupción mediante los modificadores *high_interrupt* (alta prioridad) o *low_interrupt* (baja prioridad), con lo cual se fija el bloque de código en las direcciones 0x08 o 0x018. A continuación, se muestra un ejemplo de declaración de las rutinas de interrupción:

```
void interrupt high_priority Alta_p(void)
{
  /*Ingrese el código de la rutina de servicio de
    interrupción de alta prioridad aquí:*/
}
void interrupt low_priority Baja_p(void)
{
  /*Ingrese el código de la rutina de servicio de
    interrupción de baja prioridad aquí:*/
}
```

Las funciones *Alta_p* y *Baja_p* corresponden a las rutinas de interrupción de alta y baja prioridad, respectivamente. El nombre de las funciones es irrelevante, pero se debe tomar en cuenta que toda función de interrupción debe ser de tipo void (es decir, no debe devolver parámetros) y sus argumentos también (es decir, la función no puede recibir parámetros).

En la rutina de interrupción se debe validar cada bandera de interrupción, a fin de detectar la causa de la interrupción, en el caso de que se haya activado más de una por cada nivel de prioridad. Es importante también poner las banderas de interrupción a nivel '0' antes de salir de la función.

De no utilizarse la interrupción de baja prioridad, simplemente el modificador high_priority se puede obviar y se trabajaría con una única función de interrupción.

Para comprender el funcionamiento de las rutinas de interrupción se ofrece un ejemplo práctico. En el gráfico 8.12 se muestra un diagrama esquemático del PIC18F4550 conectado a dos LED: uno por el pin RD0 y otro por el pin RD4. Además, un pulsador se conecta al pin RB0. El funcionamiento es el siguiente: cuando se detecta un flanco de bajada por el pulsador conectado al pin RB0, se activa la interrupción externa INT0, la cual estará configurada como alta prioridad y generará que el LED conectado al pin RD0 cambie de estado. Por otro lado, el desbordamiento del TMR0 generará una interrupción de baja prioridad. El prescaler del TMR0 se configura a 256 y el reloj de instrucciones trabajará a 12 MHz, con lo cual cada conteo será de 0.083 μsegundos×256, dando un tiempo de desbordamiento de 21,33 μsegundos×256=5.451 milisegundos. En la rutina de interrupción se contarán 40 desbordamientos consecutivos para cambiar el estado del LED conectado al pin RD4, con lo cual el tiempo de cada estado del será de 5,451 milisegundos×40 = 218.36 milisegundos.

En el programa 8.31 se muestra el programa completo. Se debe observar que primero se configuran los modos de prioridad de interrupción, se habilitan las interrupciones de INT0 y TMR0 y, finalmente, se habilita la máscara de interrupciones.

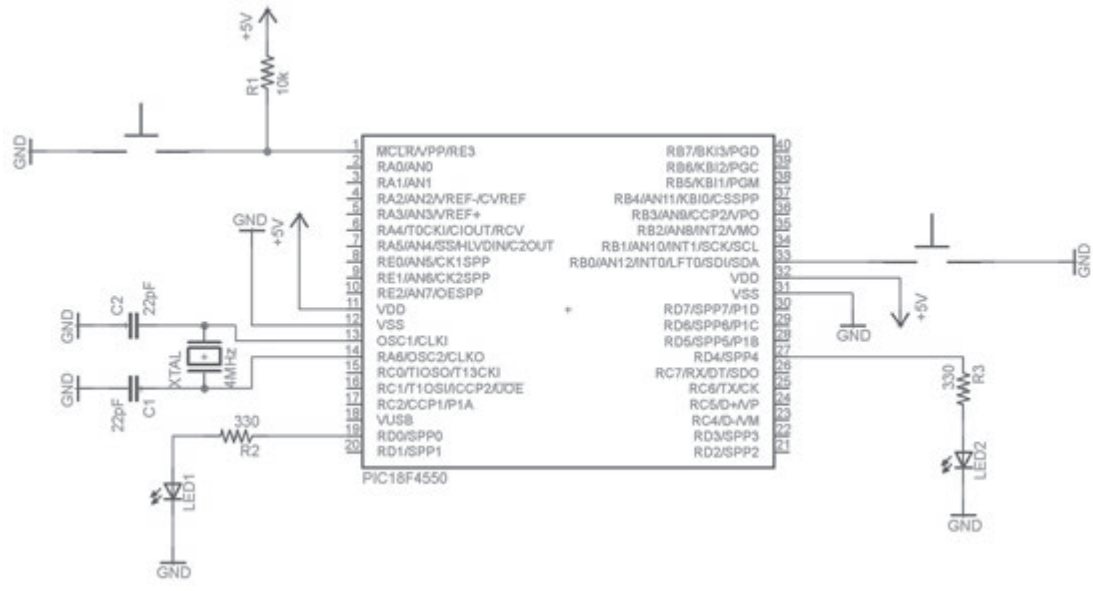
Cabe resaltar que las funciones de interrupción pueden hacer uso de variables globales, locales y estáticas sin ninguna restricción. El único cuidado que el programador debe tener es que estas rutinas sean lo más cortas posibles para no afectar a otras interrupciones que podrían ejecutarse en paralelo o para no afectar la ejecución del programa principal.

Programa 8.31. Ejemplo de configuración de una interrupción de alta y otra de baja prioridad en el lenguaje ANSI C del compilador MPLAB XC8

```
void main()
{
        TRISB = 0xFF;
        LATD=0;
        TRISDbits.TRISD0  = 0;
        TRISDbits.TRISD4  = 0;
        LATD = 0;
        RCONbits.IPEN = 1;              //Habilitando prioridades.
        INTCON2bits.INTEDG0 =0;         //Sensible al flanco de bajada.
        INTCON2bits.RBPU  = 0;          //Pull up interno del Puerto B activo.
        INTCONbits.INT0IE = 1;
        INTCONbits.GIEH   = 1;
        INTCON2bits.TMR0IP = 0;         //Baja prioridad TMR0.
        T0CONbits.T08BIT = 1;           //Contador de 8 bits.
        T0CONbits.T0CS = 0;             //Selección de la señal de reloj del CPU para el TMR0.
        T0CONbits.PSA = 0;              //Uso del Prescaler
```

```
        T0CONbits.T0PS2 = 1;
        T0CONbits.T0PS1 = 1;
        T0CONbits.T0PS0 = 1;            //Prescaler = 256
        T0CONbits.TMR0ON = 1;
        INTCONbits.TMR0IE = 1;
        INTCONbits.GIEL = 1;
        while(1);
}
void interrupt high_priority Externa(void)
{
        LATDbits.LATD0 = ~LATDbits.LATD0;
        INTCONbits.INT0IF = 0;
}
void interrupt low_priority Timer0(void)
{
        static unsigned char a = 0;
        a++;
        if(a == 10)
        {
                a = 0;
                LATDbits.LATD4 = ~LATDbits.LATD4;
        }
        INTCONbits.TMR0IF = 0;
}
```

Gráfico 8.12. Diagrama esquemático del circuito para la ejecución de dos interrupciones de alta (INT0) y baja prioridad (TMR0)



Elaboración propia.

8.12 Las directivas del preprocesador

Las directivas del preprocesador son sentencias que no son instrucciones, pero pueden cambiar el flujo del programa o permiten también mejorar su legibilidad. Por preprocesador se entiende que estas directivas son procesadas por el compilador, mas no llegan a formar parte del programa que será ejecutado en el sistema embebido. Una directiva se caracteriza por llevar por delante el símbolo # seguido de su nombre. Las directivas no requieren finalizar con punto y coma, ya que no son instrucciones. Al igual que en el lenguaje ensamblador, el ANSI C posee directivas que cumplen con funciones específicas. Estas directivas son las siguientes:

1. #define
2. #undef
3. #ifndef
4. #include
5. #if-#else-#endif
6. #ifdef
7. #error
8. #warning
9. #pragma

A continuación se explica la función de cada directiva del preprocesador y se presentan programas demostrativos que permitan entender con mayor claridad la utilidad de cada directiva.

1. La directiva #define

La directiva #define se vio anteriormente. Permite crear un alias a un valor constante y de esta manera mejorar la legibilidad del programa. Pero también es posible definir un alias a una operación o bloques de código de programa que no se desean reescribir. El alias de esta directiva normalmente se escribe en mayúsculas, con el fin de evitar confundirlas con las variables, que, por lo general, se escriben en minúsculas.

En el programa 8.32 se muestran los tres ejemplos. Primero, se define el alias UMBRAL que representa el valor decimal 10. Cuando la variable a es asignado con UMBRAL, recibe el valor 10.

El segundo alias de nombre SUMA recibe dos parámetros: *a* y *b*. El proceso equivalente a estos dos parámetros resulta ser a+b. Cabe mencionar que SUMA no es una función, ya que simplemente se está reemplazando SUMA(a,b) por la operación a+b. Cuando en el programa se ejecuta *k = SUMA(a,5)*; en realidad se ejecuta la instrucción *k = a+5*.

Finalmente, el alias RUTINA equivale a dos instrucciones consecutivas: *LATB = k*; y k++;. Véase que en la declaratoria de RUTINA la última instrucción termina en punto y coma, lo cual implica que cuando se invoque el alias en el programa no será necesario colocar otro punto y coma más.

Programa 8.32. Ejemplo de uso de la directiva #define

```
#define UMBRAL 10
#define SUMA(a,b) (a+b)
#define RUTINA  LATB = k; k++;
unsigned char a,k;
void main()
{
        TRISB = 0x00;
        a = UMBRAL;
        k = SUMA(a,5);
        RUTINA
        while(1);
}
```

2. La directiva #undef

Se tiene que en un proyecto en MPLAB X IDE, haciendo uso del compilador XC8, se han incluido diez librerías, las cuales no han sido creadas por el programador. Por lo tanto, este desconoce cuáles son los alias definidos en los archivos de cabecera. Se quiere definir el alias UMBRAL para el valor de 20. Si este alias ya ha sido definido en los archivos de cabecera, entonces el compilador arrojará un error (porque no se puede declarar el mismo alias dos veces en un mismo proyecto).

Así, el programador tiene dos opciones. Ubicar la definición del alias UMBRAL en los archivos de cabecera y comentarlo, o eliminar la definición del alias UMBRAL y volverlo a definir con el valor deseado. Esta última es la opción más adecuada, ya que no se alterará el contenido de las librerías. El programa 8.33 muestra un ejemplo.

Programa 8.33. Ejemplo de uso de la directiva #undef

```
#define UMBRAL 40
#undef UMBRAL
#define UMBRAL 70
unsgined char a;
void main()
{
    a = UMBRAL;
    while(1);
 }
```

En el ejemplo mostrado se define el alias UMBRAL con el valor de 40 en base decimal. Si se volviera a definir el mismo alias con el valor de 70, pues simplemente el compilador arrojaría un error.

Para evitar esto, se coloca la directiva #undef UMBRAL, la cual deshace toda definición previa del alias UMBRAL. En el programa mostrado la variable a toma el valor de 70.

3. La directiva #ifndef

Por otro lado, en el caso de que se deseara definir un alias con un nombre determinado si y solo si este no ha sido definido en algún archivo de las librerías agregadas al proyecto. Si el alias ha sido definido, pues se mantendrá su valor original; y de no haber sido definido, se le asignará un nuevo valor. Para lograr esta condición se requiere hacer uso de la directiva #ifndef y de la directiva #endif.

Este esquema se observa en el programa 8.34. Si el alias límite no ha sido previamente creado, entonces se crearán los alias LIMITE para el valor de 18 y NIVEL para el valor de 15, de manera que la variable *a* tendrá asignado el valor 15 y *k* el valor 18. Pero ¿qué pasaría si el alias LIMITE ya hubiera sido creado previamente? La respuesta es que el programa no compilaría, ya que el alias NIVEL no sería declarado, con lo cual el compilador no sabría qué valor asignarle a la variable *a*.

Programa 8.34. Ejemplo de funcionamiento de la directiva #ifndef

```
#ifndef LIMITE
        #define LIMITE 18
        #define NIVEL  15
#endif
unsigned char a,k;
void main()
{
        a = NIVEL;
        k = LIMITE;
        while(1);
}
```

4. La directiva #include

La directiva #include es bastante conocida en el texto. Permite incluir archivos de cabecera de extensión .h (librerías) en el proyecto. En cada archivo de este tipo de extensión se tienen declaraciones de prototipos de funciones, nuevos tipos de datos y directivas #define. Esta directiva usualmente se coloca en la parte superior de los archivos de extensión .c (que son agregados al proyecto mediante el inspector de archivos), pero también puede ir en cualquier otra ubicación.

Existen dos formatos para la directiva #include. En uno de ellos se coloca el nombre del archivo a incluir entre los símbolos <>, mientras que en el otro formato se coloca el nombre del archivo entre comillas dobles (“ ”). En el primer formato se asume que el archivo a incluir se encuentra en la carpeta de nombre *Include* dentro de todo el directorio raíz de instalación del MPLAB XC8. Por ejemplo, cuando se coloca al inicio del programa la directiva: #include <xc.h>, se sabe que este archivo es provisto por Microchip y, por tanto, se encuentra en la carpeta en mención.

Cuando se quiere incluir un archivo de cabecera creado por el programador, lo recomendable es guardarlo en la carpeta del proyecto e incluirlo entre comillas dobles. Por ejemplo, si se tiene una librería para el manejo de la pantalla LCD gráfica se puede incluir el archivo GLCD.h de la siguiente manera: #include "GLCD.h". Sin embargo, ¿qué pasaría si se tuviera una carpeta de librerías y se deseara mantener dicha carpeta en una ruta conocida del directorio raíz que no es la misma ruta del proyecto? En este caso, se deberá incluir el directorio entre las comillas dobles. Por ejemplo:

#include "C:\Librerias_Microchip\PIC18F\PIC18F4550\GLCD.h".

5. Las directivas #if-#else-#endif

En algunas ocasiones, cuando se está creando un nuevo proyecto, el programador tiene que decidir entre diferentes versiones de código. Por ejemplo, puede desarrollar una rutina que es útil en ciertas ocasiones y otra rutina que será necesaria en otras situaciones. Para evitar tener dos archivos con cada versión de código, se puede hacer uso de las directivas de compilación condicional. Estas directivas presentan un formato muy similar a las instrucciones *if-else*, con la diferencia de que en este caso aquella condición verdadera significará la compilación de un bloque de código, mas no su ejecución por parte del procesador.

Las directivas #if-#else-#endif evalúan la existencia de un alias; en función de esto, compilan un código comprendido entre #if-#else o #else-#endif. Lo que se debe entender es que la opción falsa hará que el código no seleccionado sea invisible para el compilador. En el programa 8.35 se ofrece un ejemplo.

Programa 8.35. Ejemplo de uso de las directivas #if-#else-#endif

```
#define LIMITE 45
unsigned char a,k;
void main()
{
        #if LIMITE
                TRISB = 0x00;
                a = 45;
                k = a+2;
                LATB = a+k;
        #else
                TRISB = 0x0F;
                a = 0x05;
                k = a+10
                LATB = k;
        #endif
        while(1);
}
```

Como se puede observar en el programa, se define el alias LIMITE con el valor 45 y se crean las variables *a* y *k* de 8 bits cada una. En el programa principal se evalúa la existencia de LIMITE como alias. Dado que esta condición es verdadera, se ejecutan las instrucciones comprendidas entre la directiva #if LIMITE y la directiva #else. Las demás instrucciones comprendidas entre la directiva #else y #endif no son compiladas. Tomar en cuenta que la instrucción *k = a+10* carece de punto y coma al final, ¡lo cual es un error de compilación! Sin embargo, el compilador no advierte esto, ya que se encuentra ignorando este código. Si se eliminara la definición de LIMITE, el error aparecería en la venta de salida del compilador.

6. La directiva #ifdef

La directiva #ifdef es muy parecida a la directiva #ifndef, con la diferencia de que la condición verdadera se da si el alias se encuentra definido. En el programa 8.36 se analiza un ejemplo.

Programa 8.36. Ejemplo de funcionamiento de la directiva #ifdef

```
#define LIMITE 4
#ifdef LIMITE
        #define NIVEL 10
        #define UMBRAL 11
#endif
unsigned char a,k;
void main()
{
        a = NIVEL+UMBRAL;
        k = LIMITE;
        while(1);
}
```

Como se puede observar, los alias NIVEL y UMBRAL se definen solo si LIMITE ha sido definido con cualquier valor. Como esta condición es verdadera, la variable *a* toma el valor de 21 y la variable *k* el valor de 4. Si se eliminara o comentara la definición de LIMITE, entonces aparecería un error de compilación, ya que no estarían definidos ni NIVEL ni UMBRAL.

7. La directiva #error

Cuando uno activa la compilación del programa pulsando con el ratón sobre el botón *Clean and Build Project*, el compilador empieza a verificar la sintaxis del programa, que básicamente consiste en revisar que el programador respete los lineamientos de la programación en ANSI C. Si el programa tiene o no sentido, no es problema del compilador. Sin embargo, ¿qué pasaría si para ciertas condiciones no deseadas el programador propusiera que el compilador le arrojara un error a modo de aviso, impidiendo que la compilación se complete? Esto es posible gracias a la directiva #error (mensaje).

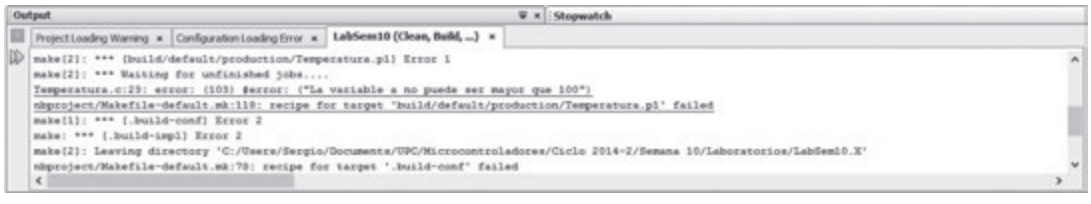
La directiva #error viene acompañada de un mensaje entre comillas dobles dentro del paréntesis. Este mensaje es el que será desplegado en la ventana de salida del compilador cuando se ejecute la directiva. Esta directiva debe ir en una instrucción condicional, de manera que, cuando se cumpla la situación no deseada, el compilador arroje el error mostrando el mensaje en la ventana de salida. Como ejemplo, se ofrece el código del programa 8.37.

Programa 8.37. Ejemplo de operación de la directiva #error

```
#define LIMITE 100
unsigned char a,k;
void main()
{
        a = 45;
        k = 20;
        k = a + k*3;
        if(k > LIMITE)
        {
                #error("La variable a no puede ser mayor que 100")
                a = 2;
        }
        else
                k = 30;
        while(1);
}
```

En el programa mostrado, la variable *a* se inicializa con el valor de 45 y *k* con 20. Luego, se realiza la asignación a *k* del valor 45+20*3 que equivale a 105. Dado que el valor de *a* supera a LIMITE (que es alias de la cifra 100), se ejecuta la directiva #error, la cual va a desplegar el error en la ventana de compilación que se muestra en el gráfico 8.13 al momento de ejecutar la opción *Clean and Build Project* en MPLAB X IDE.

Gráfico 8.13. Ventana de salida con el error de compilación debido a la directiva #error



Elaboración propia.

8. La directiva #warning

La directiva #warning opera del mismo modo que la directiva #error, con la diferencia de que no desplegará un mensaje de error en la ventana de salida y no evitará la compilación. Simplemente, aparecerá un mensaje de advertencia definido entre comillas dobles dentro del paréntesis, y la compilación seguirá su curso.

9. La directiva #pragma

Como ya se vio en los subcapítulos anteriores, la directiva #pragma permite programar los bits de configuración del microcontrolador en la cabecera del programa.

8.13 Ejemplos de aplicaciones con lenguaje ANSI C

Una vez que ya se conocen las características del lenguaje, su sintaxis, sus operadores y directivas, es posible empezar a crear código, pero para aplicaciones un poco más complejas que las vistas con el lenguaje ensamblador. La ventaja de aprender un lenguaje de alto nivel es que se simplifica la implementación de código. Es decir, con menos instrucciones y menos esfuerzo mental, es posible crear un programa más complejo.

A continuación se presentarán cuatro ejemplos aplicados que serán usados en los capítulos siguientes. Los ejemplos se presentan en orden de complejidad y los dos últimos tienen aplicación directa con dispositivos periféricos comerciales.

8.13.1 Conversión de binario a BCD

La forma más simple para realizar la conversión de binario a BCD es mediante divisiones sucesivas. Por ejemplo, la cifra 01111111_2 que representa 127 en base decimal se divide entre 100 y da como cociente 1. Luego, el residuo se divide entre 10 y se obtiene de cociente 2 y el residuo de dicha división resulta 7. Este procedimiento se puede ver en el programa 8.38, en la función bin_a_BCD.

Programa 8.38. Ejemplo de conversión de binario a BCD con divisiones sucesivas

```
void bin_a_BCD(char num);
unsigned char BCD1,BCD2,BCD3;
void main()
{
    bin_a_BCD(127);
    while(1);
}
```

```
void bin_a_BCD(char num)
{
    BCD3 = num/100;
    BCD2 = (num%100)/10;
    BCD1 = (num%100)%10;
}
```

Como se observa en el programa 8.38, para obtener un número BCD de tres cifras se requiere calcular dos divisiones y tres residuos (que al final provienen de la misma operación de división). La carga computacional detrás de estas operaciones es intensa, de manera que una rutina como bin_a_BCD llega a tomar cerca de 1457 ciclos de instrucción.

En el capítulo 5 se vio una rutina en lenguaje ensamblador para realizar la conversión de una cifra binaria de 8 bits a tres cifras BCD. Este método es más eficiente para la conversión, ya que no requiere divisiones, sino más bien desplazamientos de bit y sumas aritméticas. La función propuesta de nombre *binario_a_bcd()* se declara de un tipo compuesto por una unión y una estructura que posee cinco campos: una palabra de 2 bytes y 4 *nibbles*. En total, estos campos suman 32 bits, los cuales corresponden al campo de la unión *numero_bin*.

El número inicial se aloja en el campo *numero_ini* de 16 bits y, luego, se comienza a desplazar a la izquierda cada bit 16 veces en total. Estos desplazamientos a la izquierda afectan primero al campo nibble1, luego a nibble2, nibble3 y, finalmente, a nibble4. Conforme cada *nibble* toma un valor mayor a 4, se debe sumar el valor de 3 a dicho *nibble*, tal y como indica la regla. Al cabo de 16 iteraciones, el valor de cada *nibble* corresponde al código BCD de la cifra binaria ingresada en cuatro dígitos.

Si se ingresara a la función *binario_a_BCD* una cifra de cinco o más dígitos, pues simplemente se retornarían los cuatro dígitos BCD menos significativos del número, pero el cálculo sería erróneo. Por tal motivo, en la función se evalúa la cifra ingresada como parámetro y, si esta es mayor a 9999, se retornarán los cuatro dígitos BCD en 0.

Programa 8.39. Ejemplo de un convertidor de binario a BCD

```
typedef unión
{
        unsigned long numero_bin;
        struct{
                unsigned int numero_ini;
                unsigned char nibble1:4;
                unsigned char nibble2:4;
                unsigned char nibble3:4;
                unsigned char nibble4:4;
        }_nibbles;
}BCD;
BCD binario_a_bcd(int numero);
```

```
BCD numero;
unsigned char BCD4,BCD3,BCD2,BCD1;
void main()
{
        numero = binario_a_bcd(8221);
        BCD4 = numero._nibbles.nibble4;     //Cifra BCD correspondiente a la unidad de
                                            //millar.
        BCD3 = numero._nibbles.nibble3;     //Cifra BCD correspondiente a la unidad de
                                            //centena.
        BCD2 = numero._nibbles.nibble2;     //Cifra BCD correspondiente a la unidad de
                                            //decena.
        BCD1 = numero._nibbles.nibble1;     //Cifra BCD correspondiente a la unidad.
        while(1);
}
BCD binario_a_bcd(int numero)
{
        char i = 0;
        BCD dato;
        dato.numero_bin = 0;
        if(numero >9999) return(dato);
        dato._nibbles.numero_ini = numero;
        for(i=0;i<15;i++)
        {
                dato.numero_bin<<=1;
                if(dato._nibbles.nibble1 > 4)
                        dato._nibbles.nibble1+=3;
                if(dato._nibbles.nibble2 > 4)
                        dato._nibbles.nibble2+=3;
                if(dato._nibbles.nibble3 > 4)
                        dato._nibbles.nibble3+=3;
                if(dato._nibbles.nibble4 > 4)
                        dato._nibbles.nibble4+=3;
        }
        dato.numero_bin<<=1;
        return(dato);
}
```

Es posible diseñar la rutina de conversión de binario a BCD de manera que sea más eficiente. Un método podría ser la inclusión de la rutina de lenguaje ensamblador vista en el capítulo 5 dentro del programa en ANSI C.

8.13.2 Rutinas de retardo de tiempo

Las rutinas de retardo de tiempo son fundamentales en la programación de sistemas embebidos. En el capítulo 5 se vieron diseños de rutinas de retardo haciendo uso de los ciclos de instrucción del procesador. Sin embargo, este esquema es un poco más complicado en ANSI C, ya que la mayoría de instrucciones duran más de dos ciclos y cada una tiene una duración variable.

Para ayudar a solucionar este problema, el compilador MPLAB XC8 provee una librería de nombre delays.h que contiene cinco funciones de retardo de tiempo. Todas ellas dependen de la frecuencia de instrucciones que se esté utilizando en el programa. Estas funciones se muestran en el gráfico 8.14.

Gráfico 8.14. Funciones de retardo de la librería delays.h

Función	Descripción
Delay1TCY(void)	Retardo de un ciclo de instrucción
Delay10TCYx(unsigned char tiempo)	Retardo de 10 ciclos de instrucción multiplicados por el parámetro tiempo
Delay100TCYx(unsigned char tiempo)	Retardo de 100 ciclos de instrucción multiplicados por el parámetro tiempo
Delay1KTCYx(unsigned char tiempo)	Retardo de 1000 ciclos de instrucción multiplicados por el parámetro tiempo
Delay10KTCYx(unsigned char tiempo)	Retardo de 10,000 ciclos de instrucción multiplicados por el parámetro tiempo

Elaboración propia.

Cada función genera un retardo exacto proporcional al periodo de instrucción. De ahora en adelante, en los ejemplos se utilizará el PLL del microcontrolador con su divisor de frecuencia configurado en 1 y el divisor del CPU en 2, con lo cual se tendrá una frecuencia de reloj de 48 MHz. Dado que se requieren cuatro ciclos de reloj para ejecutar una instrucción, la frecuencia de instrucciones usada será de 12 MHz y, por tanto, el ciclo de instrucción tendrá una duración de 0.0833 µsegundos.

Entonces, la función *Delay1TCY()* generará un retardo de 0.0833 µsegundos, la función *Delay10TCYx*(unsigned char tiempo) generará un retardo proporcional a 0.833 µsegundos, ya que su retardo base es de 10 ciclos de instrucción. Por ejemplo, si el argumento tiempo es 12, entonces el retardo total será de 0.833 µsegundos × 12 = 10 µsegundos. Las demás funciones *Delay100TCYx()*, *Delay1KTCYx()* y *Delay10KTCYx()* operan con retardos base de 100 ciclos de instrucción, 1000 ciclos de instrucción y 10 000 ciclos de instrucción, respectivamente.

El parámetro tiempo de las funciones es de tipo unsigned char, lo que quiere decir que el máximo valor del multiplicador es de 255. De esta manera, el retardo más largo es *Delay10KTCYx(255)*, que corresponde a un tiempo total de 10 000×255x0.083 µsegundos = 211.65 milisegundos.

La Modulación por ancho de pulso (PWM[42]) es una técnica en la cual se modula el tiempo en alta de una señal cuadrada de periodo conocido, tal como se aprecia en el gráfico 8.15. En toda señal cuadrada periódica existen dos partes: el nivel alto y el nivel bajo. La duración de ambos niveles define el

42 Del inglés *Pulse Width Modulation.*

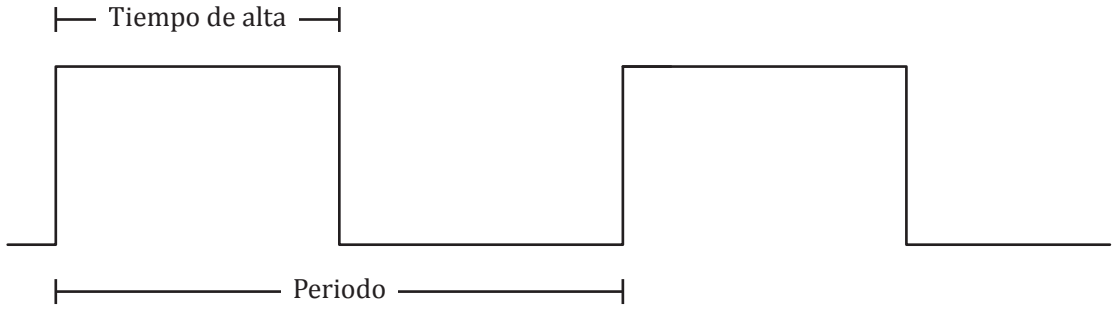
periodo de la onda. Si ambos estados son de igual duración, se dice que el ciclo de trabajo de esa señal es del 50%. Esto significa que el tiempo en alta tiene una duración equivalente a la mitad del periodo. En general el ciclo de trabajo se define de la siguiente manera:

$$Ciclo\ de\ trabajo = \frac{Tiempo\ en\ alta}{Periodo} \times 100\%$$

El PWM se utiliza en diversas aplicaciones. El uso típico es para el control de velocidad de motores, o también para el posicionamiento de servomotores. Además, existen otras aplicaciones, como conversión digital análoga o transmisión de datos, como se verá en los siguientes capítulos.

Como ejemplo, se desea generar una señal con una PWM de 10% y un periodo de 1 milisegundo. Para esto, el tiempo en alta de la señal debe tener 100 µsegundos y el tiempo en baja, 900 µsegundos. Esta señal se debe generar en el pin RD0, el cual se encuentra conectado a un circuito de interfaz a un motor de corriente continua cuya velocidad se requiere fijar a 10 RPM (con un Ciclo de trabajo del 10%). La forma de onda de la señal deseada se observa en el gráfico 8.16. En el programa 8.40 se muestra la propuesta de solución a este problema utilizando las funciones de retardo.

Gráfico 8.15. Esquema de las partes de una señal cuadrada periódica



Elaboración propia.

Programa 8.40. Generación de una señal de Ciclo de trabajo del 10%

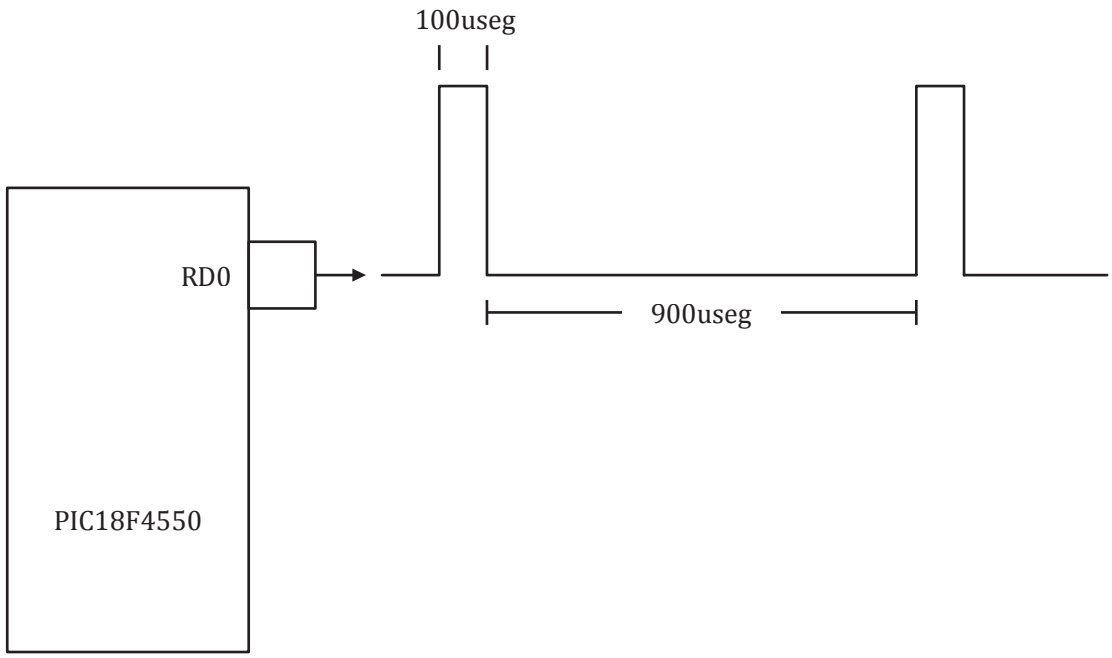
```
#include <xc.h>
#include <delays.h>
void main()
{
        LATDbits.LATD0=0;
        TRISD = 0xFE;
        while(1)
        {
                LATDbits.LATD0=1;
```

```
                Delay100TCYx(12);   //Retardo de 100 µseg
                LATDbits.LATD0=0;
                Delay100TCYx(108); //Retardo de 900 µseg
        }
}
```

El programa 8.40 muestra el código que soluciona el problema. La frecuencia de reloj se asume en 48 MHz. Cada ciclo de instrucción, entonces, tiene una duración de 0.083 µsegundos y, por tanto, la función *Delay100TCYx(12)* generará un retardo de 100×12×0.083 µsegundos = 100 µsegundos. La segunda función: *Delay100TCYx(108)* generará un retardo de 100×108×0.083 µsegundos = 900 µsegundos. Después de cada uno de los retardos, el valor del pin RD0 se actualizará con '0' para 900 µsegundos y '1' para 100 µsegundos.

En realidad, esta forma de generar la señal de PWM es muy ineficiente, ya que requiere ciclos del procesador continuos para generar los retardos de tiempo. En el próximo capítulo se mostrará un método para generar señales de PWM más preciso y que no hace uso continuo del procesador, sino más bien de la interrupción por comparación del *Timer 2*.

Gráfico 8.16. Señal de Ciclo de trabajo del 10% generada por el pin RD0



Elaboración propia.

8.13.3 Uso de la librería matemática para generar una onda sinusoidal

Una de las grandes ventajas que el compilador MPLAB XC8 presenta es su compatibilidad con algunas de las librerías estándar del ANSI C. Una de las librerías más interesantes que presenta la herramienta de desarrollo es la librería de funciones matemáticas math.h.

Hasta el momento, con lo que se ha visto, se han realizado operaciones aritméticas básicas: suma, resta, multiplicación y división. ¿Qué ocurre si se desea obtener el seno de un ángulo en radianes o el logaritmo neperiano de una cifra? ¿Es esto posible? Pues con la librería matemática se pueden hacer múltiples operaciones de este nivel. Solo se requiere declarar una variable de tipo flotante para poder operar las funciones de la librería que a continuación se enumeran en el gráfico 8.17.

Gráfico 8.17. Funciones de la librería matemática math.h

Función	Descripción
acos(float angulo)	Retorna el arco coseno de un ángulo en radianes.
asin(float angulo)	Retorna el arco seno de un ángulo en radianes.
atan(float angulo)	Retorna el arco tangente de un ángulo en radianes.
ceil(float numero)	Redondeo de la cifra al entero superior próximo.
cos(float angulo)	Retorna el coseno de un ángulo en radianes.
cosh(float angulo)	Retorna el coseno hiperbólico de un ángulo en radianes.
exp(float num)	Retorna el número neperiano elevado al valor especificado en num.
fabs(float numero)	Retorna el valor absoluto de número.
floor(float numero)	Retorna el valor de número redondeado al entero inferior más próximo.
fmod(float dividendo, float divisor)	Retorna el residuo de la división dividendo÷divisor.
log(float numero)	Retorna el logaritmo neperiano de número.
log10(float numero)	Retorna el logaritmo base 10 de número.
pow(float x, float y)	Retorna el número x elevado al exponente y.
sin(float angulo)	Retorna el seno de un ángulo en radianes.
sinh(float angulo)	Retorna el seno hiperbólico de un ángulo en radianes.
sqrt(float num)	Retorna la raíz cuadrada de num.
tan(float angulo)	Retorna la tangente de un ángulo en radianes.
tanh(float angulo)	Retorna la tangente hiperbólica de un ángulo en radianes.

Elaboración propia.

Cabe mencionar que el uso de estas funciones genera una altísima carga computacional al procesador PIC18. En general, estas funciones se pueden utilizar si es que la aplicación no presenta requerimientos de velocidad muy exigentes. Si bien es cierto que son funciones de mucha ayuda, en muchos casos prácticos estas funciones no se pueden implementar, ya que generan cargas computacionales muy altas que pueden alterar los requerimientos de tiempo de otros procesos.

Como ejemplo, el PIC18F4550 se conecta a un convertidor digital analógico de 8 bits a través del Puerto D. Se desea crear los valores binarios que dan forma a una señal sinusoidal de 500 Hz de frecuencia y una amplitud de 100 niveles con un nivel medio de 100 niveles. Para crear una onda sinusoidal se requiere tabular al menos un periodo completo de la onda y, para esto, se necesita un conjunto de valores numéricos que definan dicho periodo.

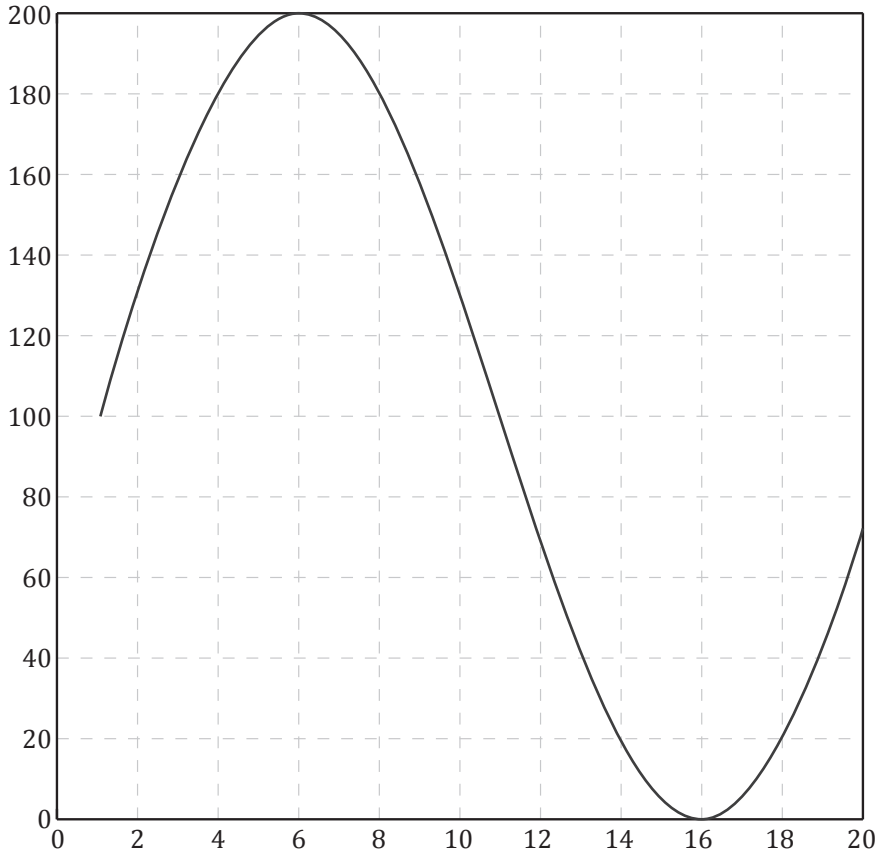
Para generar la onda se precisa definir una frecuencia de muestreo. Si la frecuencia de muestreo es de 10 kHz, esto implica que cada muestra deberá cambiar de valor en cada 100 µsegundos. De esta manera, un periodo completo de onda sinusoidal de 500 Hz de frecuencia tendrá una duración de 2 milisegundos. Para una tasa de 10 kHz, el periodo de 2 milisegundos representa 20 muestras. La forma de onda a generar se muestra en el gráfico 8.18.

Para crear los 20 puntos correspondientes a un periodo completo de 500 Hz se debe aplicar la siguiente ecuación:

$$y = 100 \times sin(2\pi \times t) + 100$$

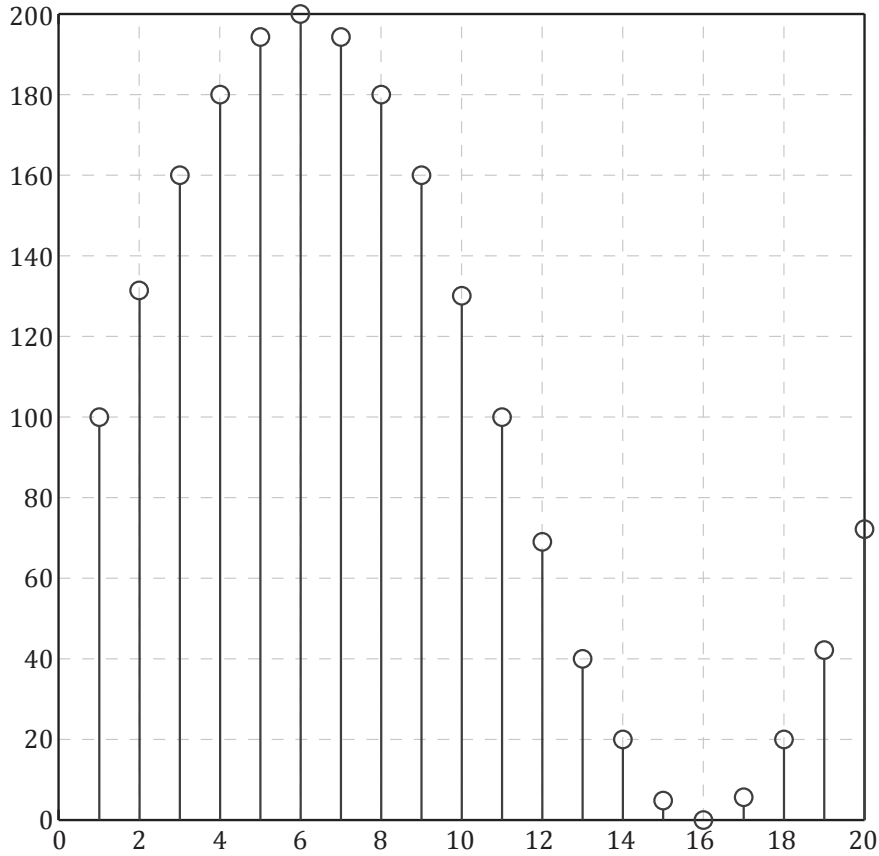
donde t = 0, 0.0001, 0.0002…..... 0.0019 segundos. Estos valores corresponden a un periodo completo para 500 Hz de frecuencia. Si se tabulan los 20 valores obtenidos en la ordenada para cada valor de t (abscisa), se tendrá la gráfica del gráfico 8.19.

Gráfico 8.18. Señal sinusoidal de 500 Hz de frecuencia



Elaboración propia.

Gráfico 8.19. Los veinte valores tabulados para generar la forma de onda sinusoidal de amplitud 100, valor medio 100 y frecuencia 500 Hz



Elaboración propia.

La idea es crear un programa que envíe cada 100 µsegundos cada uno de los veinte valores de la señal por el Puerto D. Dado que este puerto es de 8 bits, cada coordenada en el eje de las ordenadas debe estar redondeada al entero más próximo. Como la amplitud de la onda es 100 y el nivel medio también, los valores oscilarán entre 0 y 200, con lo cual este rango se puede representar con una cifra de 8 bits. En el programa 8.41 se muestra la solución a este problema haciendo uso de la función matemática *sin()*. Véase que se crea un vector de tipo unsigned char de 20 elementos y en el primer bucle iterativo *for()* se asignan los valores enteros de la variable flotante y al arreglo *vector_seno500Hz[]* haciendo una conversión de tipos mediante la instrucción: *vector_seno500Hz[i]* = (unsigned char)y. En el segundo bucle iterativo, la función *Delay1KTCYx(12)* provee los 100 µsegundos de retardo entre cada muestra. Una vez que se completa un ciclo completo de envío de 20 muestras al Puerto D, se vuelve a enviar el mismo conjunto de valores para los siguientes 2 milisegundos, con lo que se mantiene la secuencia numérica que genera la onda sinusoidal de forma indefinida.

Programa 8.41. Ejemplo de generación de una onda sinusoidal de 500 Hz de frecuencia

```
#include <xc.h>
#include <delays.h>
#include <math.h>

#define FM   10000
#define PI   3.141516
#define FREC 500
#define AMP  100
#ifdef FM
        #define DT 0.0001
#endif

float t = 0, y;
unsigned char vector_seno500Hz[20];
char i;
void main()
{
        for(i = 0;i<20; i++)        //Creación de los 20 valores que componen un periodo
                                    //completo de 500Hz.
        {
                y = AMP*sin(2*PI*FREC*t)+AMP;
                t = t+DT;
                vector_seno500Hz[i] = (unsigned char)y;
        }
        LATD  = 0;
        TRISD = 0x00;
        i = 0;
        while(1)
        {
                LATD = vector_seno500Hz[i];
                Delay1KTCYx(12);
                i++;
                if(i==20) i = 0;
        }
}
```

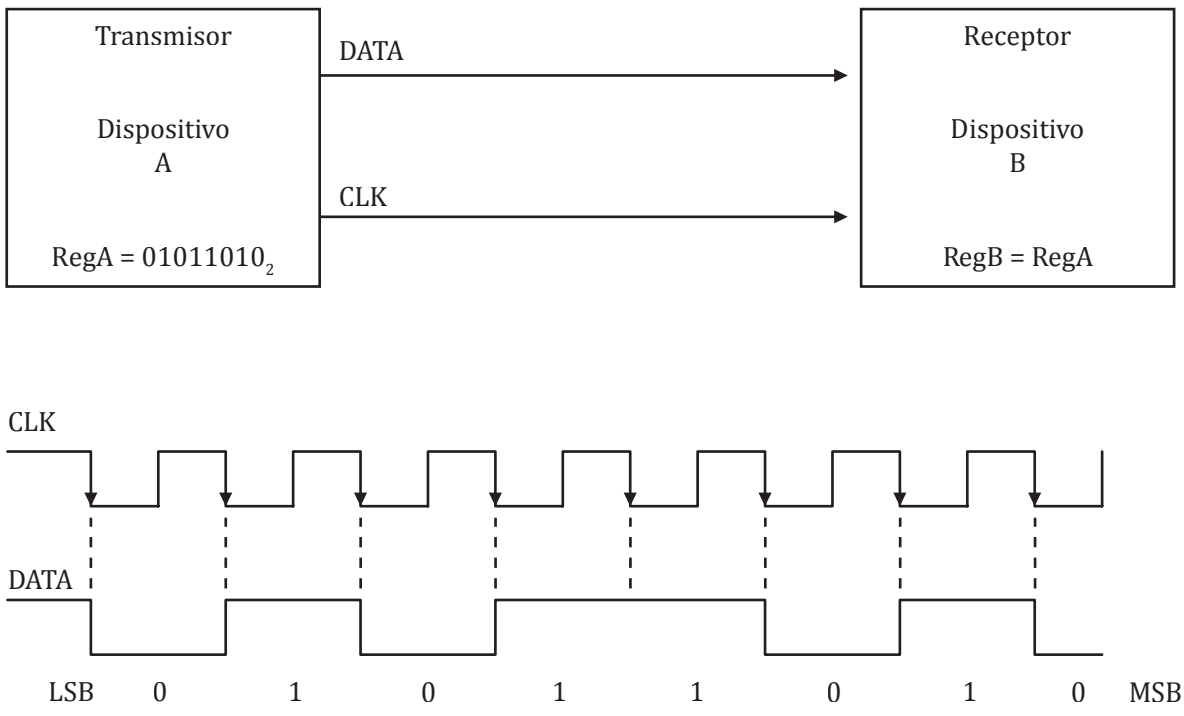
8.13.4 Lectura de un teclado PS/2

Una aplicación interesante para poner a prueba el lenguaje C y algunos de los módulos internos vistos anteriormente es una interfaz de comunicación con un teclado PS/2 usado antiguamente por las computadoras IBM compatibles.

El puerto PS/2 fue originalmente creado en el año 1987 por la empresa IBM para las conexiones de ratones y teclados a los computadores personales. Este puerto funciona de forma bidireccional para el caso del teclado y de modo unidireccional para el caso del ratón. Si bien la señalización es similar, las tramas de datos enviadas desde ambos dispositivos son diferentes. Existen otros dispositivos adicionales que usan este protocolo, como lectoras de códigos de barras y lectores de cintas magnéticas. El puerto PS/2 provee de energía a los dispositivos periféricos, por lo cual estos no requieren de una fuente externa. A pesar de esto, los dispositivos conectados a este puerto no se pueden reconectar y extraer en caliente, como el puerto USB. Cada dispositivo debe conectarse al puerto mientras el computador esté desenergizado, para que al momento de ser encendido sea reconocido por el sistema operativo y opere de forma correcta.

La comunicación usada en el protocolo PS/2 es de tipo serial síncrona. Esto significa que para el envío de un byte de información se utilizan dos líneas: datos y reloj. Por la primera línea viajan los bits del byte de información y por la segunda una señal de reloj cuyo flanco de bajada indica el momento en el cual se debe realizar la lectura de la línea de datos.

Gráfico 8.20. Protocolo serial síncrono



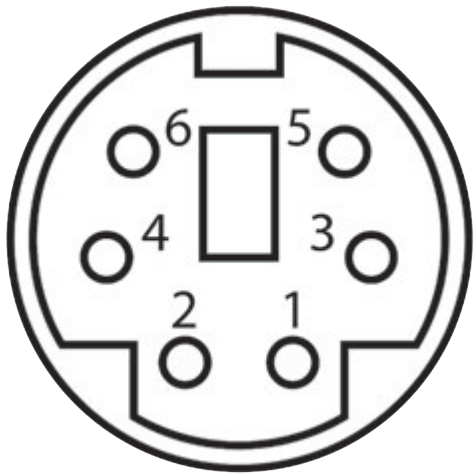
Elaboración propia.

En el gráfico 8.20 se muestra un esquema de la comunicación serial síncrona similar a la usada en el protocolo PS/2. El transmisor o dispositivo A se encarga de generar los pulsos de reloj y la señal de datos. El dispositivo B o receptor debe detectar los flancos de bajada de la señal de reloj y realizar la lectura de la línea de datos. Una transferencia se compone de 8 o más pulsos de reloj. Una vez detectado el octavo flanco, el receptor habrá completado la recepción del byte. En una transferencia en el protocolo PS/2, el bit menos significativo (LSB) se envía primero.

El puerto PS/2 del computador se compone de un conector MINI-DIN tipo hembra para el caso del computador o transmisor y macho para el caso del teclado. El conector provee la energía de 5 V necesaria para que el dispositivo periférico pueda energizarse. En el gráfico 8.21 se muestra el diagrama de pines del conector MINI-DIN hembra. La función de los pines es la siguiente:

PIN 1: Línea de datos
PIN 2: Reservado
PIN 3: Tierra (GND)
PIN 4: 5 V (con soporte máximo de 100 mA)
PIN 5: Señal de reloj
PIN 6: Reservado

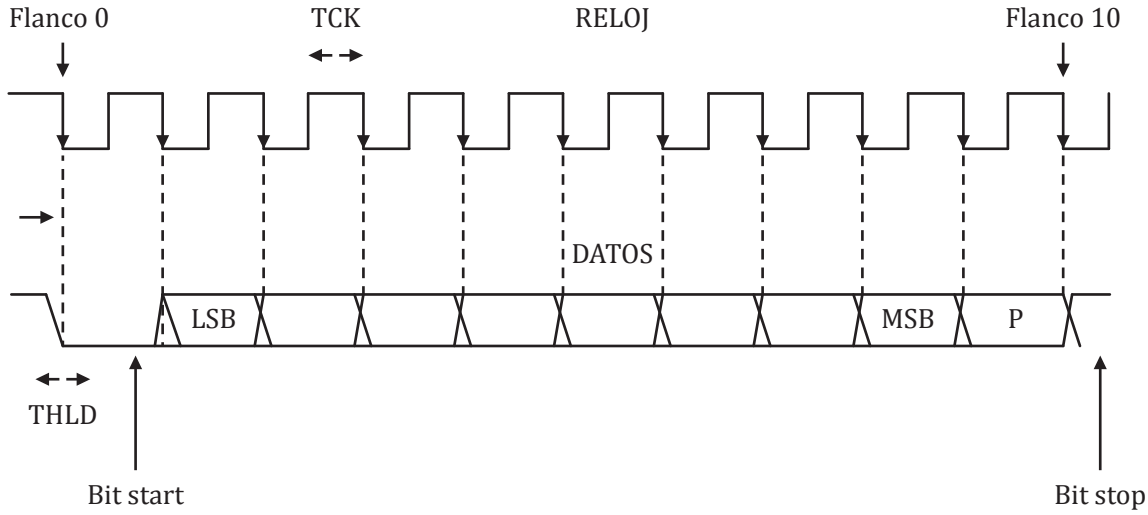
Gráfico 8.21. Conector MINI-DIN hembra



Elaboración propia.

La señal de reloj del teclado PS/2 presenta una frecuencia entre 10 a 16.7 kHz, y el estado de la señal de datos se encuentra estable antes y después de 5 µsegundos del flanco de bajada. La frecuencia de los pulsos de reloj puede variar de acuerdo con la marca del teclado usado. La trama de datos se compone de 11 bits en total. El primer bit es el bit de inicio o *Start*, el cual siempre es un nivel bajo. El siguiente bit es el LSB del byte de datos. El décimo bit es el bit de paridad, que sirve para detectar errores en la trama, y se encuentra configurado en modo impar (quiere decir que, si el número de bits con nivel '1' de la trama de datos es impar, el bit de paridad será '0'). Finalmente, se tiene el bit de parada o *Stop*, que siempre se mantiene en nivel '1'. En el gráfico 8.22 se muestra una imagen de las señales de reloj y datos para el protocolo PS/2.

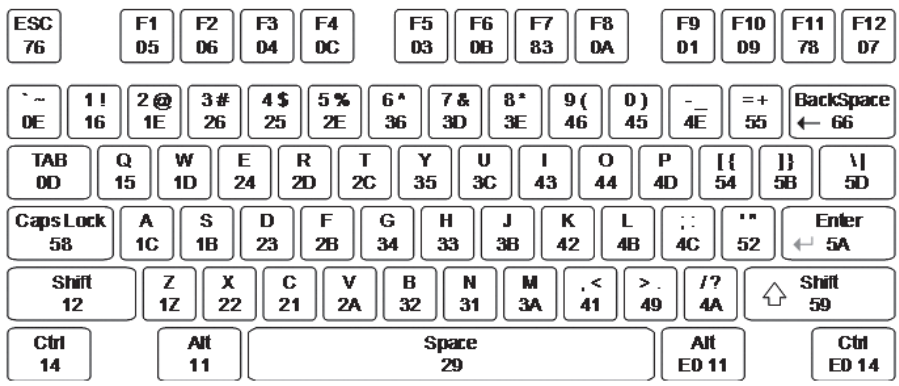
Gráfico 8.22. Señales de datos y reloj en el protocolo PS/2



Elaboración propia.

Las señales de reloj y datos por defecto se encuentran en nivel alto cuando no hay actividad en el teclado, esto significa que no se ha pulsado ninguna tecla. Una vez que se pulsa una tecla, el controlador del teclado enviará una secuencia de bytes correspondientes al código de la tecla pulsada y al estado de la pulsación. El código de la tecla pulsada se denomina *Scan Code* y es un código binario especial para este tipo de teclado matricial. Mientras el usuario mantenga la tecla pulsada, el controlador de teclado enviará continuamente el *Scan Code* cada 100 milisegundos. Cuando el usuario suelte la tecla, se enviará un código de tecla levantada conocido como *Key Up Code*, el cual se representa por el byte 0xF0. Este código se enviará seguido del *Scan Code*. Algunas teclas se conocen como «extendidas», lo que implica que el *Scan Code* viene precedido de otro byte con el valor 0xE0. Entre estas teclas se tienen los cursores *Alt* y *Ctrl*. En el gráfico 8.23 se observa un diagrama de un teclado PS/2 con la distribución de teclas estándar, y debajo de cada tecla se muestra su *Scan Code*.

Gráfico 8.23. Diagrama de distribución de teclas de un teclado PS/2 y sus *Scan Codes*



Fuente: http://www.soliddepot.com/forum/viewtopic.php?f=9&t=21

Para diseñar un sistema de detección de teclas pulsadas del teclado PS/2 se hará uso del PIC18F4550 y el módulo de interrupción externa INT0. En el gráfico 8.24 se muestra el diagrama de conexiones del teclado al PIC18F4550. La señal de reloj se conecta al pin RB0/INT0. Entonces, la interrupción externa 0 se configura con flanco de bajada; por cada uno de los 11 pulsos de reloj se obtendrá una interrupción.

Dentro de la rutina de interrupción se validará el orden del bit y si se hará una conversión serial a paralelo haciendo uso de una unión. Se detectará el orden de llegada de cada bit, empezando del segundo al noveno, donde se encuentran los bits de datos. Una vez que se han completado los 11 flancos de bajada correspondientes a un byte de datos, se actualiza la variable código con el *Scan Code* o *Key Up Code* y se ingresa a una rutina donde se hace uso de este valor. En el programa 8.42 se muestra un ejemplo de programación que permite detectar el valor del *Scan Code* en una variable compuesta de tipo dato_PS2.

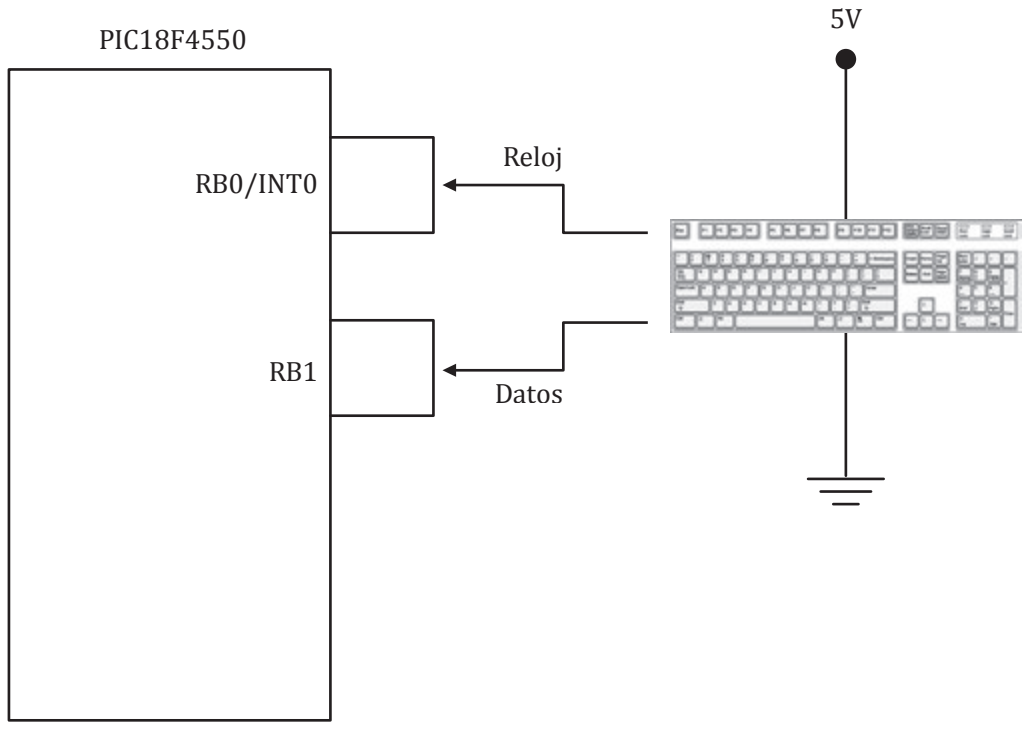
Programa 8.42. Detección de teclas pulsadas proveniente de un teclado PS/2

```
typedef union{
        unsigned char _Byte_PS2;
        struct{
                char bit0: 1;
                char bit1: 1;
                char bit2: 1;
                char bit3: 1;
                char bit4: 1;
                char bit5: 1;
                char bit6: 1;
                char bit7: 1;
        }_bits;
}dato_PS2;
dato_PS2 dato;
char codigo = 0;
void main()
{
  TRISB                 = 0xFF;
  RCONbits.IPEN         = 0;    //Sin prioridades.
  INTCON2bits.INTEDG0   = 0;    //Sensible al flanco de bajada.
  INTCON2bits.RBPU      = 0;
  INTCONbits.INT0IE     = 1;
  INTCONbits.GIE        = 1;
  while(1)
  {
        if(codigo != 0)
        {
```

```
            /*La rutina de análisis de la tecla pulsada
           puede ir aquí. Esta rutina debe analizar el
             ScanCode o Key Up Code contenido en la variable
             codigo*/
          codigo = 0;
            }
    }
}
void interrupt RUT_INT0(void)
{
            static char i = 0;
            if(i==1)            dato._bits.bit0 = PORTBbits.RB1;
            else if(i==2)       dato._bits.bit1 = PORTBbits.RB1;
            else if(i==3)       dato._bits.bit2 = PORTBbits.RB1;
            else if(i==4)       dato._bits.bit3 = PORTBbits.RB1;
            else if(i==5)       dato._bits.bit4 = PORTBbits.RB1;
            else if(i==6)       dato._bits.bit5 = PORTBbits.RB1;
            else if(i==7)       dato._bits.bit6 = PORTBbits.RB1;
            else if(i==8)       dato._bits.bit7 = PORTBbits.RB1;
            i++;
            if(i==11)
            {
                        i = 0;
                        codigo = dato._Byte_PS2;
            }
            INTCONbits.INT0IF = 0;
}
```

Queda como tarea para el lector completar el código de análisis de la tecla pulsada para transformar el *Scan Code* en un código ASCII. De la misma manera, la rutina de interrupción se puede completar para validar el bit de paridad impar que se encuentra en la posición i == 10 y generar un código de error en caso de que se detecte una incompatibilidad. Para esto, la rutina deberá contar el número de bits en estado '1' y verificar si esta cantidad es par o impar. Si es impar, el bit de paridad deberá ser '0'; de ser par, este bit será '1'.

Gráfico 8.24. Diagrama de conexión del teclado PS/2 al PIC18F4550 para la propuesta de solución



Elaboración propia.

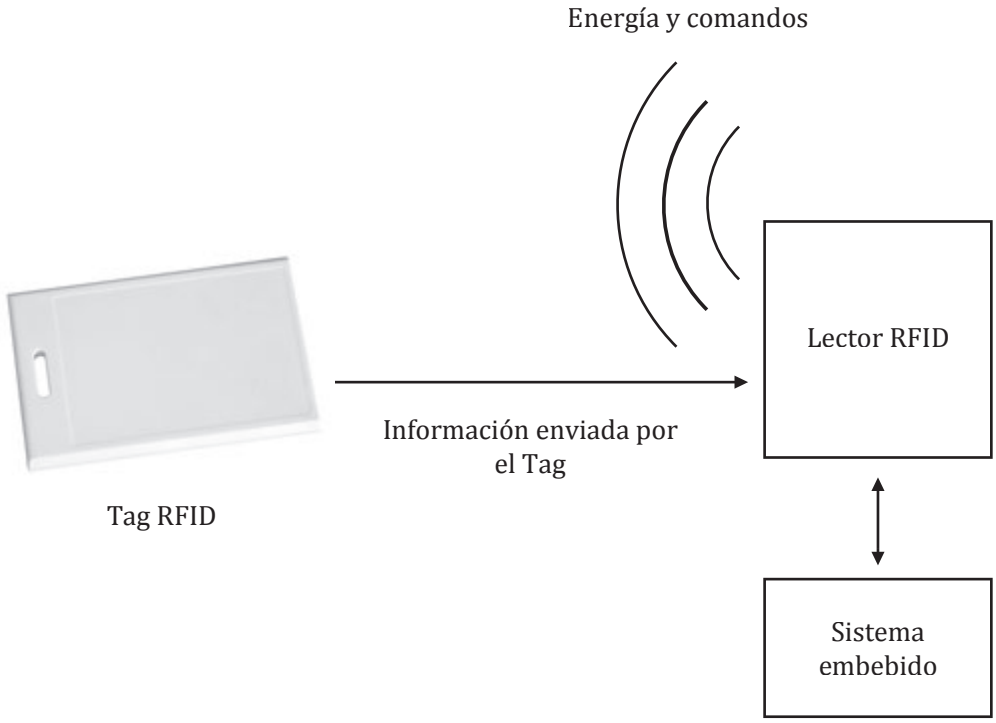
8.13.5 Lectura de códigos de tarjetas magnéticas mediante un lector RFID

RFID (Identificación por radiofrecuencia o *Radio Frequency Identificaton*, por sus siglas en inglés) es una tecnología que permite obtener códigos binarios de etiquetas ubicadas a cierta distancia de un lector, el cual emite ondas de radiofrecuencia. Un sistema básico de RFID se compone de una tarjeta o *Tag*, un lector y un computador o sistema embebido encargado de reconocer el código de la tarjeta. El lector puede recibir información del *Tag*; a su vez, también puede grabar en esta nueva información. Uno de los ejemplos más conocidos de sistemas RFID es el que se encuentra en la entrada a la estación de buses Metropolitano de la ciudad de Lima. El pago del pasaje y el acceso a los terminales se realiza mediante *Tags* RFID y lectores conectados al sistema de acceso al terminal. Cuando el sistema confirma el saldo disponible en la tarjeta, el usuario tiene derecho al ingreso (mediante una barda que afloja el torque permitiendo el acceso de una persona) y el descuento del saldo es automáticamente descargado de la tarjeta. En el gráfico 8.25 se observa un esquema de un sistema RFID. El *Tag* RFID se encuentra a una cierta distancia del lector, que es de unos cuantos centímetros para el tipo de *Tag* mostrado en la figura (*Tag* pasivo).

Para obtener una respuesta del *Tag* RFID, el lector emite una onda de radiofrecuencia que irradia a la tarjeta, si es que esta se encuentra dentro del rango de emanación de energía o rango de lectura del lector (usualmente alrededor de unos 10 centímetros). De esta manera, el *Tag* utiliza esta energía para retornar su propio código de identificación al lector. Tanto los lectores como los *Tags* utilizan antenas de tipo *Microstrip*, que corresponden a un diseño de antena adherido al circuito impreso, con lo cual ocupan muy poco espacio y pueden estar implementadas en tarjetas, etiquetas flexibles o hasta en

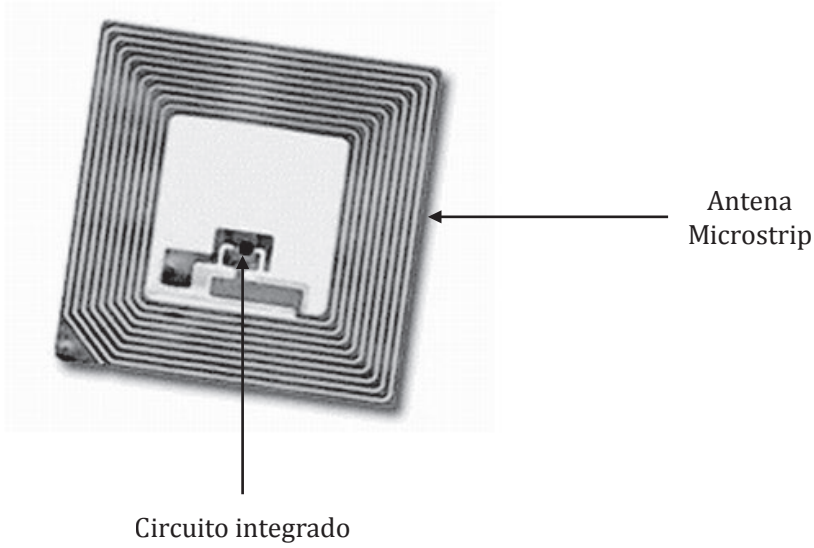
llaveros. Adicionalmente, el *Tag* cuenta con un circuito integrado muy pequeño, el cual es el encargado de generar la señal de información a enviar y controlar una pequeña memoria EEPROM donde se almacenará el código del *Tag* [17]. En el gráfico 8.26 se puede ver una imagen de un Tag RFID[43].

Gráfico 8.25. Comunicación entre un lector RFID y un Tag pasivo



Elaboración propia.

Gráfico 8.26. Arquitectura de un Tag RFID



Elaboración propia.

43 Cfr. Gidekel 2006.

Como se puede observar, el tamaño del *Tag* depende en gran medida del diseño de la antena. Estas son creadas en base a cobre, aluminio e inclusive a tintas conductoras. El tamaño y diseño de la antena define su rango de lectura. El tipo de tarjeta mostrada en el gráfico 8.26 corresponde al tipo pasivo, ya que recibe la energía de alimentación a través de la señal de radiofrecuencia generada por el lector. Existen otros tipos de *Tags*, denominados «activos», los cuales tienen un rango de transmisión mucho mayor, con la desventaja de que requieren su propia fuente de alimentación.

La banda de frecuencia para la cual operan los lectores y *Tags* depende del tipo de aplicación en la cual se utilizará el sistema RFID. Por ejemplo, la banda LF (Baja frecuencia, del inglés *Low Frequency*) opera en el rango de los 125 kHz. En esta frecuencia, la comunicación es lenta y el alcance es corto (no supera los 50 centímetros). Además, no se permite la lectura de dos *Tags* simultáneamente, por lo cual esta banda es usualmente aplicada en sistemas de marcación de personal o el acceso a sistemas de seguridad. La banda de los 13.56 MHz, denominada HF (Alta frecuencia, del inglés *High Frequency*), presenta una mayor velocidad de comunicación y tiene un rango de lectura de alrededor de un metro. Se utiliza en librerías, contenedores y tarjetas inteligentes. La tercera banda en UHF se encuentra en el rango de los 868 a 928 MHz, y con ella se logran rangos de lectura de 9 metros y se pueden leer hasta 1200 *Tags* simultáneamente. Finalmente, la banda de 2.4 a 5.8 GHz presenta una baja velocidad de transmisión y un rango de lectura menor que 2 metros. Entre sus aplicaciones se encuentran los sistemas de telepeaje.

La empresa MikroElektronika presenta entre su línea de productos diversos dispositivos periféricos para la comunicación con sistemas embebidos. Entre estos productos se encuentra un lector RFID basado en el circuito integrado EM4095, el cual es un transceptor RFID que se conecta y polariza una antena *Microstrip* a una frecuencia de 125 kHz, y que tiene un bus de comunicación de cuatro hilos para recibir señales de control y enviar datos al sistema embebido. En el gráfico 8.27 se muestra una imagen del módulo RFID. Cada *Tag* puede almacenar hasta 64 bits de información.

Gráfico 8.27. Lector RFID basado en el transceptor EM4095 de MikroElektronika

Fuente: http://www.mikroe.com/add-on-boards/various/rfid-reader/

El EM4095 se alimenta con un voltaje máximo de 7 V. Este circuito integrado se encarga de manejar todo el proceso de comunicación inalámbrico con el *Tag* pasivo y se comunica con un sistema embebido a través de cuatro líneas, cuyas funciones se muestran en el gráfico 8.28.

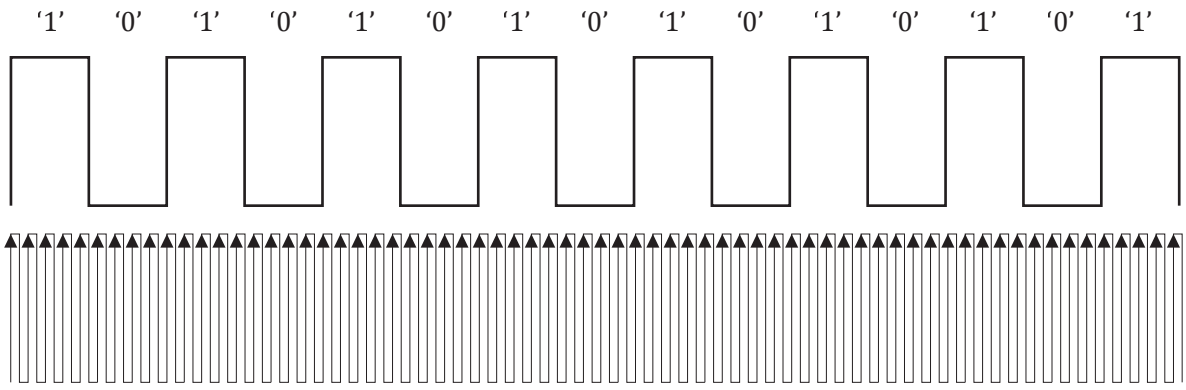
Gráfico 8.28. Función de los pines de comunicación del EM4095

Pin	Función
RDY/CLK	Esta señal de salida contiene la frecuencia de reloj de datos de aproximadamente 120 kHz. Este pin, además, opera como bandera de que el integrado se encuentra en modo de bajo consumo (nivel '0').
DEMOD_OUT	Es la señal de datos del EM4095. Contiene el código binario leído desde el *Tag*.
SHD	Es un pin de entrada usado para mantener el EM4095 en bajo consumo (con nivel '1'). En este modo no es posible leer la información del *Tag*.
MOD	Cuando este pin recibe un nivel '1', se reinicia la modulación de señal en la antena.

Elaboración propia.

Conforme el *Tag* se aproxima a la antena del lector, las señales RDY/CLK y DEMOD_OUT se mantendrán generando el código binario leído desde la tarjeta. Es importante que la tarjeta se aproxime mucho a la antena del lector para que la calidad de ambas señales mejore. Las señales aparecerán siempre que las líneas SHD y MOD se encuentren en nivel '0'. El código de un *Tag* RFID comienza con nueve transiciones de '1' a '0'. Esta etapa permite sincronizar al receptor para detectar el código de la tarjeta RFID y sigue la norma EM4100, en que se deben leer solamente los bits impares (en realidad, la secuencia de sincronismo es 111111111_2, los '0's se deben ignorar, al igual que el resto de bits que mantienen la misma posición par en la secuencia). En el gráfico 8.29 se muestra una imagen de la señal DEMOD_OUT y la señal de reloj en la etapa de sincronismo. Los flancos de subida de la señal determinan el momento en el cual la señal de datos cambia de estado; por tanto, son indicadores para el sistema embebido del momento en el que deberá leer el bit de la línea DEMOD_OUT.

Gráfico 8.29. Bloque de sincronismo para ubicar el código del *Tag* RFID

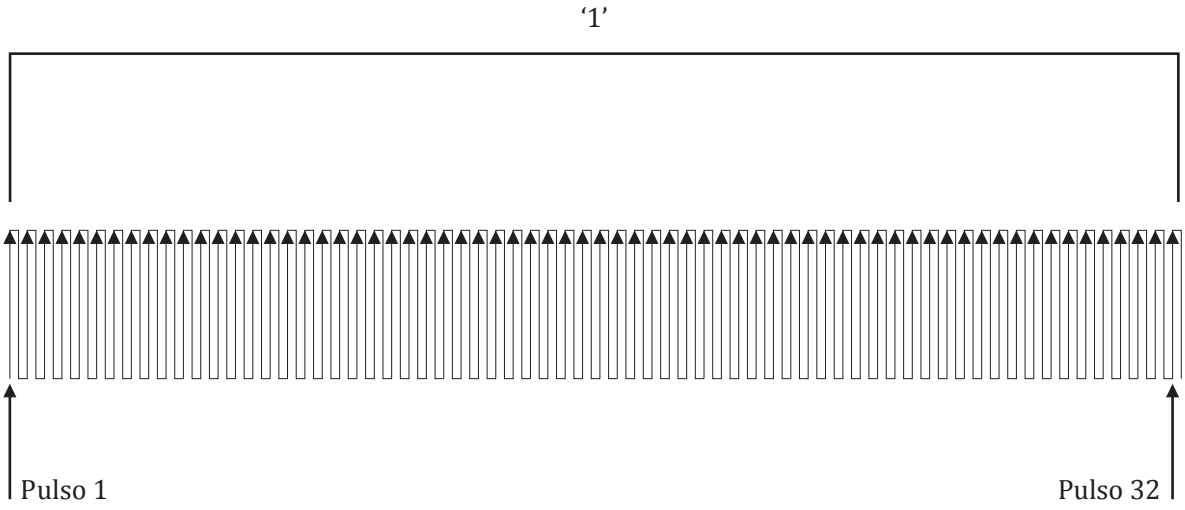


Elaboración propia.

El periodo de la señal de reloj es de aproximadamente 8.4 µsegundos, y el tiempo de bit en la línea es de aproximadamente 268.8 µsegundos. Esto significa que la duración de un bit en la línea DEMOD_OUT es de 32 periodos de reloj. El cambio de estado de la línea de datos se sincroniza con uno de los flancos de subida de la señal de reloj, pero solo uno de cada 32 será el flanco que determine el cambio de estado. En el gráfico 8.30 se muestra una duración de un bit '1' en la línea DEMOD_OUT. Se puede observar que el tiempo de bit se encuentra compuesto por 32 periodos de reloj.

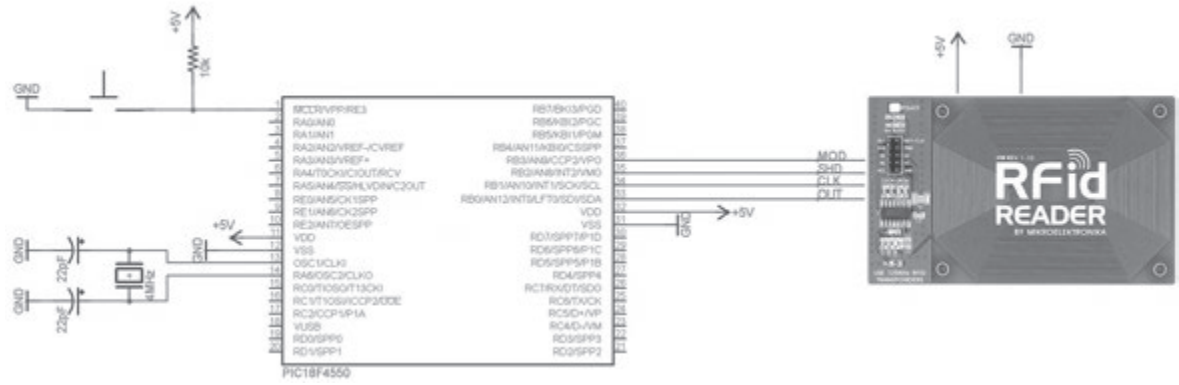
La propuesta de solución a este problema se presenta en el gráfico 8.31. La línea DEMO_OUT se conecta al pin RB0, mientras que la señal de reloj se conecta al pin RB1. Ambos pines poseen la característica de poder ser configurados como entradas de interrupción externa. La línea RB2 controla la señal SHD y la línea RB3 la señal MOD. Estos pines se configuran como salidas digitales y se mantienen en nivel '0'.

Gráfico 8.30. Duración en ciclos de reloj de un bit en la línea DEMOD_OUT



Elaboración propia.

Gráfico 8.31. Circuito esquemático propuesto para la lectura del código de los *Tags* RFID



Elaboración propia.

La propuesta consiste en detectar el flanco de subida de la señal de datos por la interrupción externa INT0/RB0. Una vez detectada esta interrupción, se deberá verificar que este bit sea '1' y corresponda al primer elemento de la transición de nueve '1's y '0's. Para esto, se realizarán los siguientes pasos:

1. Configurar la interrupción externa INT0 sensible al flanco de subida.
2. Configurar la interrupción externa INT1 sensible al flanco de subida.
3. Activar las interrupciones e inicializar el contador de flancos *cnt* = 0. La variable de sincronismo *one_seq* se inicializará en 0.
4. En cada interrupción por flanco de subida del INT1 se deberá incrementar en 1 la variable *cnt*.
5. En cada interrupción por flanco de subida en INT0 se deberá consultar si *cnt* es mayor o igual a 63 o menor o igual a 65. Esto implica que hay dos flancos de subida consecutivos de la señal de datos en un periodo igual 64 periodos de reloj aproximadamente (con un error de tolerancia de ±1 periodo de reloj). Si la condición es falsa, no se ha detectado una transición de '1' a '0', por lo cual la variable one_seq se inicializará en 0. En caso de que la condición sea verdadera, se incrementará en 1 la variable *one_seq* y luego se consultará si esta variable tomó el valor 9. Finalmente, el contador cnt se inicializará en 0 para que se reinicie el conteo de los pulsos de reloj en el siguiente tiempo de bit de la señal DEMOD_OUT.
6. Una vez detectado el pulso de sincronismo, se desactivará la interrupción en INT0 y se inicializará *cnt* en 0. Posteriormente, se esperará a que cnt sea 16 (la mitad de un tiempo de bit) y se comenzará a leer el valor de la línea RB0 cada vez que cnt sea 32. Luego, la variable cnt se inicializará en 0 y se repetirá el proceso durante 25 o más ciclos (dependiendo de la cantidad de bits que se quiera extraer de la tarjeta).

Cada tarjeta posee un código binario único. El estándar EM4100 establece los campos a los que pertenece cada bit, los bits de paridad y el final de la trama. El programa 8.43 muestra un ejemplo para extraer los primeros 25 bits de la trama en un vector. Se recomienda al lector diseñar una función que implemente el estándar EM4100 y evalúe el contenido en datos de los bits obtenidos del *Tag* para determinar si hubo o no error de lectura. Un detalle muy importante sobre el lector RFID de MikroElektronika es que los *Tags* deben aproximarse lo máximo posible a la antena del lector a fin de que la calidad de la señal sea buena y, por tanto, la detección de los bits sea correcta.

Programa 8.43. Algoritmo para extraer los primeros 25 bits de código de un *Tag* RFID

```
#include <xc.h>
#include <delays.h>

#pragma config FOSC      = XTPLL_XT     //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV    = 1            //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV    = OSC1_PLL2    //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT      = ON
#pragma config WDT       = OFF
#pragma config LVP       = OFF
#pragma config BOR       = OFF
#pragma config DEBUG     = ON
#pragma config VREGEN    = OFF
#pragma config PBADEN    = OFF
#pragma config MCLRE     = ON

#define OUT              PORTBbits.RB0
#define RDY_CLK          PORTBbits.RB1
#define SHD              LATBbits.LATB2
#define MOD              LATBbits.LATB3

#define OUT_Direction    TRISBbits.TRISB0
#define RDY_CLK_Direction TRISBbits.TRISB1
#define SHD_Direction    TRISBbits.TRISB2
#define MOD_Direction    TRISBbits.TRISB3

#define  CLAVE_1   1
#define  CLAVE_2   2
#define  CLAVE_3   3
#define BUSCA_SYNC      0
#define ESPERA_MEDIO_BIT 1
#define OBTIENE_BITS     2
#define MOSTRAR_BITS     3
#define REINICIO         4
#define INICIO           5

char data[256]@0x100;
unsigned char sync_flag,one_seq,i;
unsigned char cnt,clave;
char k1,k2,k3 = 0;
unsigned char estado = INICIO;
```

```
char clave_1[]= "11001000001001000000000010";
char clave_2[]= "11001000001001001000010100";
char clave_3[]= "11000010001000011000010000";

void main()
{
        OUT_Direction = 1;
        RDY_CLK_Direction = 1;
        SHD_Direction = 0;
        MOD_Direction = 0;
        SHD = 1;
        MOD = 0;
        TRISD = 0x00;
        TRISC = 0b10111111;
        INTCON2bits.RBPU    = 0;
        INTCON2bits.INTEDG0 = 1; //Flanco de subida en RB0
        INTCON2bits.INTEDG1 = 1; //Flanco de subida en RB1
        INTCONbits.INT0IF   = 0;
        INTCON3bits.INT1IF  = 0;
        INTCONbits.PEIE     = 1;
        INTCONbits.GIE      = 1;
        SHD = 0;
        while(1)
        {
                switch(estado)
                {
                        case INICIO:
                                Delay10KTCYx(240);     //Tiempo de espera para leer el
                                                       //código del siguiente Tag una
                                Delay10KTCYx(240);     //vez que se ha detectado
                                Delay10KTCYx(240);     //correctamente un código anterior.
                                INTCONbits.INT0IE  = 1;
                                INTCON3bits.INT1IE = 1;
                                estado = BUSCA_SYNC;
                        case BUSCA_SYNC:
                                if(one_seq == 9)
                                {
                                        one_seq = 0;
                                        INTCONbits.INT0IE = 0;
                                        estado = ESPERA_MEDIO_BIT;
                                }
                        break;
```

```
                        case ESPERA_MEDIO_BIT:
                                while(cnt!=16);
                                cnt = 0;
                                i=1;
                                data[0]=OUT;
                                estado = OBTIENE_BITS;
                        break;
                        case OBTIENE_BITS:
                                while(cnt!=32);
                                cnt = 0;
                                data[i]= OUT;
                                i++;
                                if(i==128) estado = MOSTRAR_BITS;
                        break;
                        case MOSTRAR_BITS:
                                k1=0;
                                k2=0;
                                k3=0;
                                for(i=0;i<25;i++)
                                {
                                        if(data[i]==clave_1[i]-'0') k1++;
                                        if(data[i]==clave_2[i]-'0') k2++;
                                        if(data[i]==clave_3[i]-'0') k3++;
                                }
                                if(k1==25) clave = CLAVE_1;
                                if(k2==25) clave = CLAVE_2;
                                if(k3==25) clave = CLAVE_3;
                                estado = INICIO;
                        break;
                }
        }
}

void interrupt RUT_INT(void)
{
        if(INTCON3bits.INT1IF==1)       //Entrada del clock
        {
                cnt++;
                INTCON3bits.INT1IF=0;
        }
        if(INTCONbits.INT0IF==1)        //Señal de data
        {
```

```
                if(cnt>=63 && cnt <=65)
                {
                        one_seq++;
                }
                else
                {
                        one_seq = 0;
                }
                cnt = 0;
                INTCONbits.INT0IF = 0;

        }
}
```

Video

Control de 4 displays 7 segmentos de tipo ánodo común cuyos cátodos se encuentran conectados al puerto D del PIC18F4550.

http://bit.ly/1VNvWDQ

Resolución de videos en HD

Preguntas de repaso

1. Analice el siguiente código:

```
unsigned char secuencia[] = "45678903210389127654";
unsigned char i,j;
unsigned char mn,save,aux;
void main()
{
                PORTB = 0;
                TRISB = 0;
                for (i = 0; i<20;i++)
                {
```

```
                        mn = '9'+ 1;
                        for (j=i;j<20;j++)
                        {
                                if(secuencia[j]<mn) {mn = secuencia[j]; save=j;}
                        }
                        aux = secuencia[i];
                        secuencia[i]=mn;
                        secuencia[save]=aux;
                        PORTB = save;
                }
        while(1);
}
```

¿Qué función cumple el programa sobre el vector *secuencia[]*?

2. Diseñe una función de nombre *num_unos* que retorne el número de 1_2 que contiene el parámetro de entrada a la función que es un valor numérico de 16 bits. En el ejemplo mostrado en la función main(), la variable *num* debería recibir el valor de 11:

```
char num_unos(unsigned int valor);
void main()
{
        char num;
        num = num_unos(0b1011101110110101);
        while(1);
}
```

3. Diseñe una función que retorne el valor de la raíz cuadrada de un número de 16 bits. Si el parámetro de entrada a la función no tiene raíz cuadrada exacta, la función deberá retornar 0. En el ejemplo mostrado, la función *raíz_cuadrada(900)* debe retornar el valor de 30 a la variable *num*:

```
#define NO_TIENE_RAIZ 0
unsigned int raiz_cuadrada(unsigned int num);
unsigned int num;
void main()
{
        num = raiz_cuadrada(900);
        while(1);
}
```

4. Analice el siguiente código en ANSI C:

```
char mensaje[] = "Hola UPC Monterrico.";
typedef union{
                struct{
                        char _byte_LSB;
                        char _byte_MSB;
                }_bytes;
                int _dbyte;
}tipo_byte;
tipo_byte dato;
char *p;

void main()
{
        p = &mensaje[0];
        p+= 5;
        *p+=2;
        dato._bytes._byte_LSB = p[5];
        dato._bytes._byte_MSB = *p;
        dato._dbyte++;
        while(1);
}
```

Responda las siguientes preguntas:

a. Al finalizar el programa, ¿qué valor toma la variable *dato._bytes._byte_LSB*?
b. Al finalizar el programa, ¿qué valor toma la variable *dato._bytes._byte_MSB*?
c. Al finalizar el programa, ¿qué valor toma la variable *p[10*]?
d. Al finalizar el programa, ¿el contenido del vector mensaje ha cambiado?

5. Analice el siguiente código:

```
#define ESTADO1 0
#define ESTADO2 1
#define ESTADO3 2
#define ESTADO4 3
char estado;
char a,b;
void main()
{
```

```
        a = 3; b = 4;
        estado = ESTADO1;
        while(1)
        {
                switch(estado)
                {
                        case ESTADO1:
                                a++;
                                estado = ESTADO2;
                                break;
                        case ESTADO2:
                                b++;
                                a--;
                                estado = ESTADO4;
                                break;
                        case ESTADO3:
                                b = a+2;
                                a = b-1;
                                if(a>5)  estado = ESTADO4+ESTADO3;
                                else    estado = ESTADO4;
                                break;
                        case ESTADO4:
                                b --;
                                a = b;
                                estado = ESTADO3;
                                break;
                        default:
                                a = b + 2;
                                while(1);
                }
        }
}
```

Responda las siguientes preguntas:

a. Al finalizar el programa, ¿qué valor toma la variable *b*?
b. Al finalizar el programa, ¿qué valor toma la variable *a*?
c. Al finalizar el programa, ¿qué valor toma la variable *estado*?

Respuestas

Respuestas del capítulo 8:

http://bit.ly/1NnQpxv

Capítulo 9. Los módulos de temporización

En todo sistema embebido, los módulos de temporización son necesarios para garantizar con muy alta precisión los requerimientos temporales de cualquier aplicación. Por ejemplo, un reloj en tiempo real necesita una señal de reloj precisa de 1 segundo. Un generador de PWM para un servomotor requiere establecer un periodo de 20 milisegundos con suma exactitud. El cumplimiento de estos requerimientos sería prácticamente imposible sin la ayuda de los módulos de temporización.

Un módulo de temporización es simplemente un registro contador automático capaz de realizar una cuenta en un tiempo Δt. Conociendo este valor temporal y el estado de la cuenta, es posible determinar el tiempo transcurrido de un determinado evento. El hecho de que el conteo sea automático libera al procesador de la tarea de llevar la cuenta y, al mismo tiempo, generar retardos inherentes debido al acceso de lectura y escritura de la información.

Los temporizadores normalmente pueden provocar interrupciones en dos formas: por desbordamiento o por comparación. Para el primer caso, se tiene un registro de 8 bits que es capaz de autoincrementarse. Este registro comienza en el valor 0x00 y se incrementa después de un tiempo Δt en 0x01. Después de un tiempo total de 255×Δt, la cuenta habrá alcanzado el valor 0xFF. Una vez que transcurra otro ciclo Δt adicional, el contador se habrá desbordado tomando el valor 0x00. El desbordamiento puede generar que una bandera se active y, a su vez, pueda producir una interrupción cada 256×Δt.

Por otro lado, en la comparación, se carga un registro comparador con un valor; por ejemplo, 50. Si el contador comienza a contar y llega al valor en mención, una bandera se pondrá a nivel '1', con lo cual también se podrá producir una interrupción. Este evento se repetirá cada 50×Δt.

En un temporizador también es fundamental saber configurar el tiempo de conteo Δt. Por lo general, este valor depende de la frecuencia de reloj del procesador, pero también es posible hacer uso de una señal de reloj externa colocada en uno de los pines del microcontrolador o de un cristal externo de baja frecuencia o de una señal de reloj interna específica para el módulo temporizador. Los temporizadores cuentan con un Prescaler, que es un circuito divisor de frecuencia que permite adecuar la señal de entrada al valor Δt deseado.

9.1 Los temporizadores del PIC18F4550

El PIC18F4550 cuenta con cuatro temporizadores de tamaño variable (tres de ellos de 16 bits y uno de 8 bits). Cada temporizador genera su propia interrupción y realiza un conteo cada tiempo Δt programable. Adicionalmente al conteo, que es un proceso que todos los temporizadores tienen en común, algunos temporizadores colaboran con el funcionamiento de otros módulos periféricos. Por ejemplo, los temporizadores *Timer* 1 y 3 operan en conjunto con el módulo de entrada de captura, mientras que el temporizador *Timer* 2 permite generar señales con PWM.

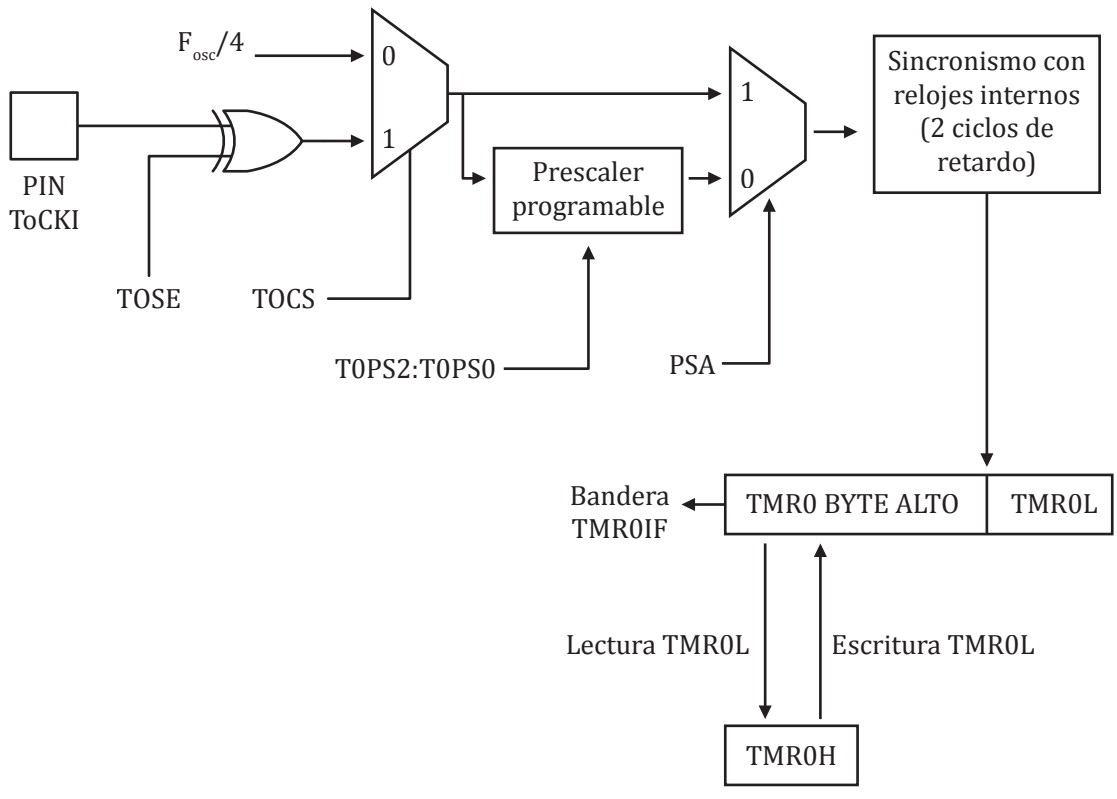
9.1.1 El *Timer* 0

El *Timer* 0 es un temporizador que puede operar en modo de 8 y 16 bits y que genera una condición de bandera o una interrupción cada vez que la cuenta se desborda. Este módulo trabaja con un divisor de frecuencia programable denominado Prescaler, el cual permite seleccionar el valor Δt de conteo. Además, puede utilizar como fuente de reloj para el conteo la señal de reloj interna del CPU, o también una señal externa conectada al pin ToCKI que corresponde a la línea RA4.

En el gráfico 9.1 se muestra un diagrama de bloques del temporizador TMR0. Allí se puede observar que la entrada de reloj del sistema ingresa a un multiplexor de 2 a 1 que permite seleccionar entre la señal de reloj de instrucciones y una señal externa conectada al pin ToCKI. Para seleccionar la señal externa, el bit TOCS que controla el primer multiplexor debe estar a nivel '1'. Esta etapa define si el registro TMR0 se configura como contador o como temporizador. Para el primer caso, se utiliza la señal externa, y para el segundo, la frecuencia de reloj interna.

El bit TOSE se debe configurar en nivel '0' o '1' para seleccionar el flanco activo que dominará el modo de conteo. En modo de temporización, el flanco activo siempre es de subida. Posteriormente, el segundo multiplexor de 2 a 1 controlado por el bit PSA definirá si la señal de reloj utilizará un divisor o ingresará con su misma frecuencia a la etapa de sincronismo, la cual genera un retardo de dos ciclos de reloj. Finalmente, esta señal controlará el conteo en el registro TMR0 de 16 bits, que se compone de dos registros de 8 bits: TMR0L y TMR0H. El registro TMR0L se actualiza automáticamente por cada flanco activo de la señal de reloj, mientras que el registro TMR0H es un registro Latch, el cual requiere de un acceso de lectura o escritura para refrescar el valor en el byte alto de TMR0 (algo parecido a la relación entre el registro PCL con los registros PCLATH y PLATU).

Gráfico 9.1. Diagrama de bloques del módulo temporizador TMR0



Elaboración propia.

El control del *Timer* 0 se realiza mediante el registro de función especial de nombre T0CON, el cual presenta 8 bits, de los cuales cada uno de ellos participa en la configuración del módulo. Este registro se muestra en el gráfico 9.2. La función de los bits es la siguiente:

El bit TMR0ON controla el encendido o apagado del TMR0. Si este bit se encuentra en nivel '1', el registro TMR0 empezará el conteo automático, pero mientras este bit se encuentre en nivel '0', el conteo estará desactivado.

Gráfico 9.2. El registro T0CON

TMR0ON	T08BIT	TOCS	TOSE	PSA	T0PS2	T0PS1	T0PS0

Elaboración propia.

El bit T08BIT permite controlar el modo de conteo del registro TMR0. Si este bit se mantiene en '1', el TMR0 operará como un solo registro de 8 bits, generando un desbordamiento cada 256 cuentas. El acceso al registro TMR0 será equivalente a acceder a TMR0L. Por otro lado, si este bit se configura en nivel '0' se configurará el modo de conteo a 16 bits, generándose un desbordamiento cada 65 536 cuentas. En esta condición requiere el uso de los registros TMR0H y TMR0L para el conteo.

El bit TOCS permite seleccionar la fuente de reloj para el conteo del TMR0. Si este bit se pone a nivel '0', se selecciona el reloj del CPU (es decir, la frecuencia de reloj generado por el cristal dividido entre 4). En este modo el módulo TMR0 trabaja como temporizador. Si el bit, se pone a nivel '1', se selecciona la señal externa conectada al pin ToCKI. Aquí el módulo TMR0 trabajará como contador, con lo cual cada cuenta representa un periodo de la señal externa.

El bit TOSE solo presenta funcionalidad si es que el bit TOCS es '1'. Este bit controla el flanco activo, que generará un incremento en el conteo del registro TMR0L. Si el bit TOSE es '1' lógico, el incremento se realizará en el flanco de bajada de la señal conectada al pin ToCKI. Si el bit es '0', entonces el incremento se realizará en el flanco de subida de la señal conectada al pin ToCKI.

El bit PSA permite seleccionar el uso del Prescaler sobre la señal. Si el bit es '1', el Prescaler queda desactivado y la señal llega al sincronizador tal como ingresa. Si el bit PSA es '0', se habilitará el Prescaler y la señal de entrada presentará una disminución en su frecuencia proporcional a un factor constante y programable.

Si se selecciona el Prescaler con el bit PSA en '0', los bits T0PS2, T0PS1 y T0PS0 controlarán el factor de división de la señal, el cual es múltiplo de la potencia de dos. En el gráfico 9.3 se muestran los divisores que pueden ser seleccionados por estos bits.

Gráfico 9.3. Divisores de frecuencia programables para el Prescaler del módulo TMR0

T0PS2	T0PS1	T0PS0	Prescaler
0	0	0	2
0	0	1	4
0	1	0	8
0	1	1	16
1	0	0	32
1	0	1	64
1	1	0	128
1	1	1	256

Elaboración propia.

El desbordamiento del TMR0 genera que un bit de bandera se ponga a nivel '1' automáticamente. Este bit de nombre TMR0IF se encuentra dentro del registro INTCON. Además, es posible que este evento genere una interrupción cuando el TMR0 pasa de 0xFF a 0x00 en modo 8 bits o de 0xFFFF a 0x0000 en modo de 16 bits. Para esto, el habilitador de interrupción TMR0IE del registro INTCON debe estar en nivel '1'. Es muy importante que durante la rutina de servicio de interrupción TMR0IF se ponga a nivel '0, para desactivar la bandera para el siguiente desbordamiento y la interrupción no se vuelva a ejecutar de manera inmediata.

9.1.2 El *Timer* 1

El temporizador *Timer* 1 es un contador binario que incluye las siguientes características:

1. Puede trabajar como contador en modo de 8 y 16 bits.
2. El conteo se realiza automáticamente en el registro TMR1, que se divide en dos registros de 8 bits: TMR1H (parte alta del contador) y TMR1L (parte baja del contador). Ambos registros son de lectura y escritura, por lo cual se puede evaluar y modificar el valor de conteo mediante instrucciones de acceso a los registros.
3. La fuente de reloj para el conteo del temporizador puede ser una señal externa o la señal de reloj de instrucciones interno. También es posible utilizar una señal de un cristal externo de baja frecuencia conectado a los pines RC0 y RC1 como fuente de reloj.
4. Presenta una interrupción por desbordamiento del TMR1.

El diagrama de bloques del *Timer* 1 se muestra en el gráfico 9.4. Allí se puede observar que se utilizan los pines RC0 (T1OSO/T13CKI) y RC1 (T1OSI) para la conexión de una señal externa que puede ser un pulso cuadrado que se conecta a la línea RC0 o un cristal de baja frecuencia conectado a los pines RC0 y RC1. El bit T1OSCEN permite seleccionar si se va a utilizar la señal de reloj externa o el cristal de baja frecuencia. El conteo del TMR1 se sincroniza con los flancos de subida de la señal de entrada externa.

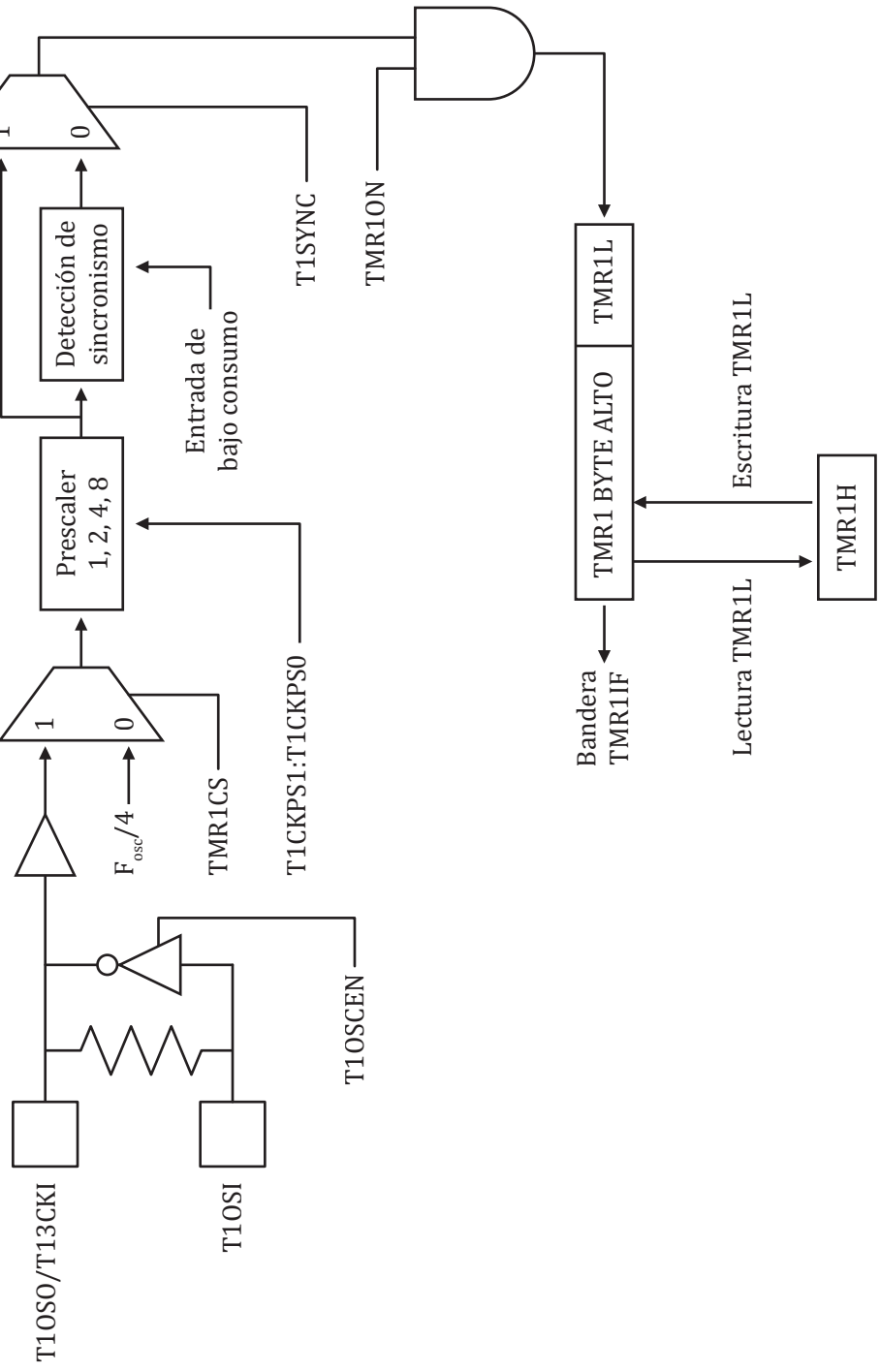
Un multiplexor controlado por el bit TMR1CS del registro T1CON permite seleccionar como fuente de reloj la señal externa o la frecuencia de instrucciones del microprocesador. Cuando se habilita la señal externa con TMR1CS, los pines RC0 y RC1 trabajan automáticamente como entradas digitales, muy independientemente de la configuración del registro TRISC. La salida del multiplexor se conecta a un Prescaler que divide la frecuencia de la señal en un factor que puede ser 1, 2, 4 u 8. Este factor se selecciona con los bits T1CKPS1 y T1CKPS0.

La salida del Prescaler se conecta a un bloque de detección de sincronismo, que es usado para despertar al microprocesador del modo de bajo consumo cuando se detecta un flanco activo de entrada. De no utilizar este modo, el bit T1SYNC se carga con '1' y la señal de salida del Prescaler se conecta directamente a una de las entradas de una compuerta AND. Dicha compuerta tiene como su otra entrada al bit TMR1ON, que, de ser '0', impedirá que la señal llegue al registro TMR1, de manera que no habrá conteo. Si este bit se pone a nivel '1', se habilitará el paso de la señal de reloj al registro TMR1, con lo cual se activará el conteo automático.

El registro TMR1 se divide en una parte alta (los 8 bits más significativos) y una parte baja (el registro TMR1L). La parte baja se puede leer y escribir directamente, pero la parte alta se accede de forma indirecta a través de un registro LATCH de nombre TMR1H. Una lectura sobre el registro TMR1L automáticamente cargará el valor alto de TMR1 en TMR1H. Por otro lado, una escritura de TMR1L automáticamente cargará el valor alto de TMR1 con el valor almacenado en TMR1H.

El control y la configuración del temporizador *Timer* 1 se realiza mediante el registro de la función especial T1CON que se muestra en el gráfico 9.5. Allí se puede observar cada uno de los 8 bits que permiten su configuración.

Gráfico 9.4. Diagrama de bloques del *Timer* 1



Elaboración propia.

Gráfico 9.5. Bits del registro T1CON

RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON

Elaboración propia.

El bit RD16 permite definir si el acceso a los registros contadores TMR1H y TMR1L se realizará en una sola operación de 16 bits (nivel '1') o en dos operaciones de lectura de 8 bits (nivel '0').

El bit T1RUN posibilita la selección de la opción del oscilador externo, que se puede utilizar para el diseño de un reloj en tiempo real. Cuando este bit es '0', se selecciona cualquiera de las fuentes de reloj externa o interna presentadas. Si el bit es '1', se selecciona el modo de reloj externo de baja frecuencia y de bajo consumo.

Los bits T1CKPS1 y T1CKPS0 permiten seleccionar el valor del Prescaler. De acuerdo con la combinación binaria, se configura un divisor. Esto se muestra en el gráfico 9.6.

Gráfico 9.6. Configuración del divisor del Prescaler para el *Timer* 1

T1CKPS1	T1CKPS0	Prescaler
0	0	1
0	1	2
1	0	4
1	1	8

Elaboración propia.

Por otra parte, el bit T1OSCEN permite activar el uso del modo de cristal externo de baja frecuencia. Si este bit se pone a nivel '1', se habilita el modo de oscilador externo. En caso contrario, queda desactivado y se debe elegir entre la señal de reloj externa o la frecuencia de instrucciones.

El bit T1SYNC permite configurar el sincronismo con la señal externa si es que el bit TMR1CS es '1'. Si el bit T1SYNC es '0', la señal de entrada se sincronizará con el modo de bajo consumo. Si este bit es '1', se deshabilitará el sincronismo.

Luego, el bit TMR1CS permite seleccionar la fuente de reloj. Si este bit se pone a nivel '0', se seleccionará como fuente de reloj la frecuencia de instrucciones. En caso contrario, se elegirá el reloj externo.

Finalmente, el bit TMR1ON permite activar el conteo del registro TMR1 dejando que la señal de reloj se conecte al contador. Este bit se pone a nivel '1' para cumplir con esta función.

Cuando el registro TMR1 se desborda (pasa de 0xFFFF a 0x0000), el bit TMR1IF se pone a nivel '1', indicando que el evento de desbordamiento ha ocurrido. Este bit se encuentra en la posición LSB dentro del registro PIR1. Es muy importante que este bit se ponga a nivel '0' una vez que se ha detectado el desbordamiento, ya que esta bandera no se inicializa de forma automática. Si se desea activar una interrupción debido a este evento, entonces se deberá activar el bit TMR1IE del registro PIE1, cargando en este un nivel '1'.

Una de las funciones especiales del temporizador *Timer* 1 es la posibilidad de conectar un cristal externo a los pines RC0 y RC1 cuyo valor es de 32 768 kHz, tal como se muestra en el gráfico 9.7. La gran ventaja de poder usar esta frecuencia de oscilación es que resulta ser un múltiplo de una potencia de 2. Si se configura el Prescaler a un valor de 2; entonces, cada vez que el TMR1 se desborde, habrán transcurrido dos segundos. Si se quiere generar una interrupción cada segundo, se pueden realizar los siguientes pasos:

1. Cargar el registro TMR1 con el valor 32 768 o 0x8000.
2. Configurar el TMR1 para que opere con el oscilador externo de baja frecuencia.
3. Activar la interrupción por el TMR1 con TMR1IE = '1'.
4. Esperar a que ocurra la interrupción cuando el bit TMR1IF sea '1'.
5. En la rutina de servicio de interrupción se debe borrar la bandera TMR1IF y volver a cargar el registro TMR1 con 0x8000. Esto permitirá que la interrupción se genere exactamente cada segundo.

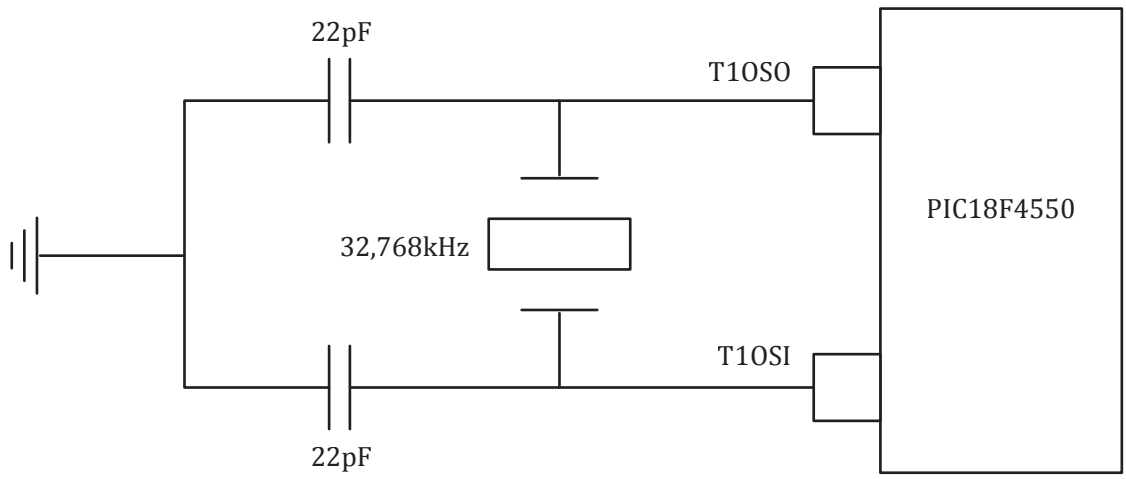
En el programa 9.1 se muestra un ejemplo de configuración de la interrupción por desbordamiento del TMR1. En este ejemplo, cada segundo se incrementa el valor del Puerto D configurado como salida.

Programa 9.1. Ejemplo de configuración del *Timer* 1 con fuente de reloj externo de 32 768 kHz

```
        #include <P18F4550.INC>
        org  0x00
        goto MAIN
        org  0x08
        goto RUT_INT
        org  0x018
        retfie
        org  0x20
MAIN:
        clrf  LATD
        clrf  TRISD
        movlw 0x80
        movwf TMR1H
        clrf  TMR1L
        movlw b'00001111'     ;Se selecciona el oscilador de 32,768kHz
        movwf T1CON           ;como fuente de reloj.
        bsf   PIE1, TMR1IE    ;Se activa la interrupción por desbordamiento
        bsf   INTCON,PEIE     ;del TMR1.
        bsf   INTCON,GIE
LAZO:
        goto  LAZO            ;Bucle indefinido
```

```
RUT_INT:
        btfss PIR1,TMR1IF
        retfie
        bcf   PIR1,TMR1IF
        bsf   TMR1H,7
        clrf  TMR1L
        incf  LATD
        retfie
        END
```

Gráfico 9.7. Conexión de un cristal externo a los pines de entrada de oscilador externo del *Timer* 1



Elaboración propia.

9.1.3 El *Timer* 2

El *Timer* 2 es un temporizador de 8 bits que no activa banderas o genera una interrupción por desbordamiento, sino que más bien trabaja mediante modo de comparación. El valor de comparación se carga en un registro de función especial de 8 bits de nombre PR2. Cuando el valor de la cuenta de TMR2 iguala al valor almacenado en PR2, la bandera TMR2IF del registro PIR1 se pone a nivel '1'. Este evento también puede generar una interrupción si es que el bit TMR2IE del registro PIE1 se pone a nivel '1'.

Adicionalmente a la función de comparación, el módulo de temporización TMR2 permite generar el periodo y ancho de pulso de una señal PWM en el módulo CCP[44]. Además, contribuye con la señal de reloj para el módulo MSSP[45] que se verá más adelante, en los capítulos finales.

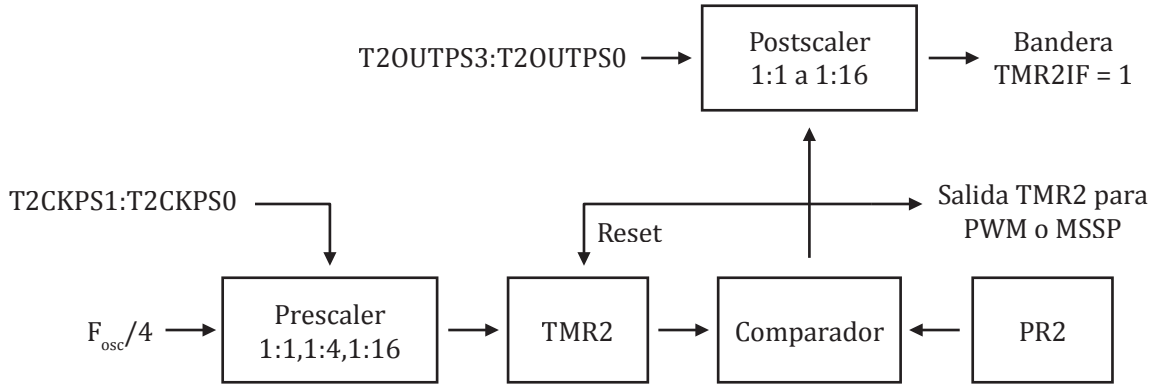
En el gráfico 9.8 se muestra el diagrama de bloques para el módulo de temporización TMR2. Como se observa, la única entrada de reloj al sistema es la frecuencia de instrucciones usada para el

44 Módulo de captura y comparación.

45 Módulo serial síncrono en modo SPI que se verá en el capítulo 12.

CPU. Esta señal se conecta a un Prescaler con divisores programables de 1, 4 y 16. Luego, la frecuencia de salida define el conteo automático del registro TMR2 de 8 bits.

Gráfico 9.8. Diagrama de bloques del módulo temporizador TMR2



Elaboración propia.

Un Postscaler permite generar divisores programables desde 1 hasta 16 para reducir la velocidad de comparación entre TMR2 y PR2. Sin embargo, el conteo de TMR2 se realiza a una velocidad mayor, que puede ser aprovechada por los módulos CCP y MSSP. Ambos registros son de lectura y escritura y pueden ser modificados en cualquier momento.

El registro de función especial T2CON permite configurar el modo de funcionamiento del TMR2. Este registro básicamente permite configurar el Prescaler, el Postscaler y el encendido del módulo. En el gráfico 9.9 se muestran los 7 bits de configuración que componen este registro. A continuación, se explica la función de sus bits.

Gráfico 9.9. El registro de configuración T2CON

--	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0

Elaboración propia.

Los bits T2OUTPS3-T2OUTPS0 permiten seleccionar el divisor del Postscaler. En total son 16 valores diferentes para la división de la frecuencia de comparación. En el gráfico 9.10 se observan los valores.

Gráfico 9.10. Configuración del divisor del Postscaler del TMR2

T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	Postscaler
0	0	0	0	1
0	0	0	1	2
0	0	1	0	3
0	0	1	1	4
0	1	0	0	5
0	1	0	1	6
0	1	1	0	7
0	1	1	1	8
1	0	0	0	9
1	0	0	1	10
1	0	1	0	11
1	0	1	1	12
1	1	0	0	13
1	1	0	1	14
1	1	1	0	15
1	1	1	1	16

Elaboración propia.

El bit TMR2ON permite habilitar o deshabilitar el conteo en el TMR2. Si este bit es '1', el módulo de temporización queda activado, mientras que si este bit es '0', entonces el conteo quedará inhabilitado.

Finalmente, los bits T2CKPS1 y T2CKPS0 permiten configurar el divisor del Prescaler. En el gráfico 9.11 se muestra la configuración de estos bits para cada uno de los tres divisores.

Gráfico 9.11. Configuración del divisor del Prescaler del TMR2

T2CKPS1	T2CKPS0	Prescaler
0	0	1
0	1	4
1	0	16
1	1	16

Elaboración propia.

Cabe resaltar que ante un evento de Reset el registro TMR2 siempre tomará el valor 0x00, mientras que PR2 lo hará en 0xFF.

9.1.4 El *Timer* 3

El temporizador TMR3 opera de un modo muy similar al TMR1. En el gráfico 9.12 se puede observar el diagrama de bloques de este módulo. Como se puede notar, la similitud con el diagrama de bloques del TMR1 (gráfico 9.4) es muy evidente. Ambos módulos comparten las mismas señales de reloj externas conectadas a los pines RC0 y RC1. Inclusive, el valor del Prescaler es similar.

El control del módulo TMR3 se realiza mediante el registro de función especial T3CON, el cual se muestra en el gráfico 9.13 con sus bits de configuración. A continuación se explica la función de cada bit.

El bit RD16 permite configurar el acceso al registro TMR3 en una sola operación de 16 bits cuando está a nivel '1'. Por otro lado, si este bit se pone a nivel '0', el acceso será en dos operaciones de 8 bits.

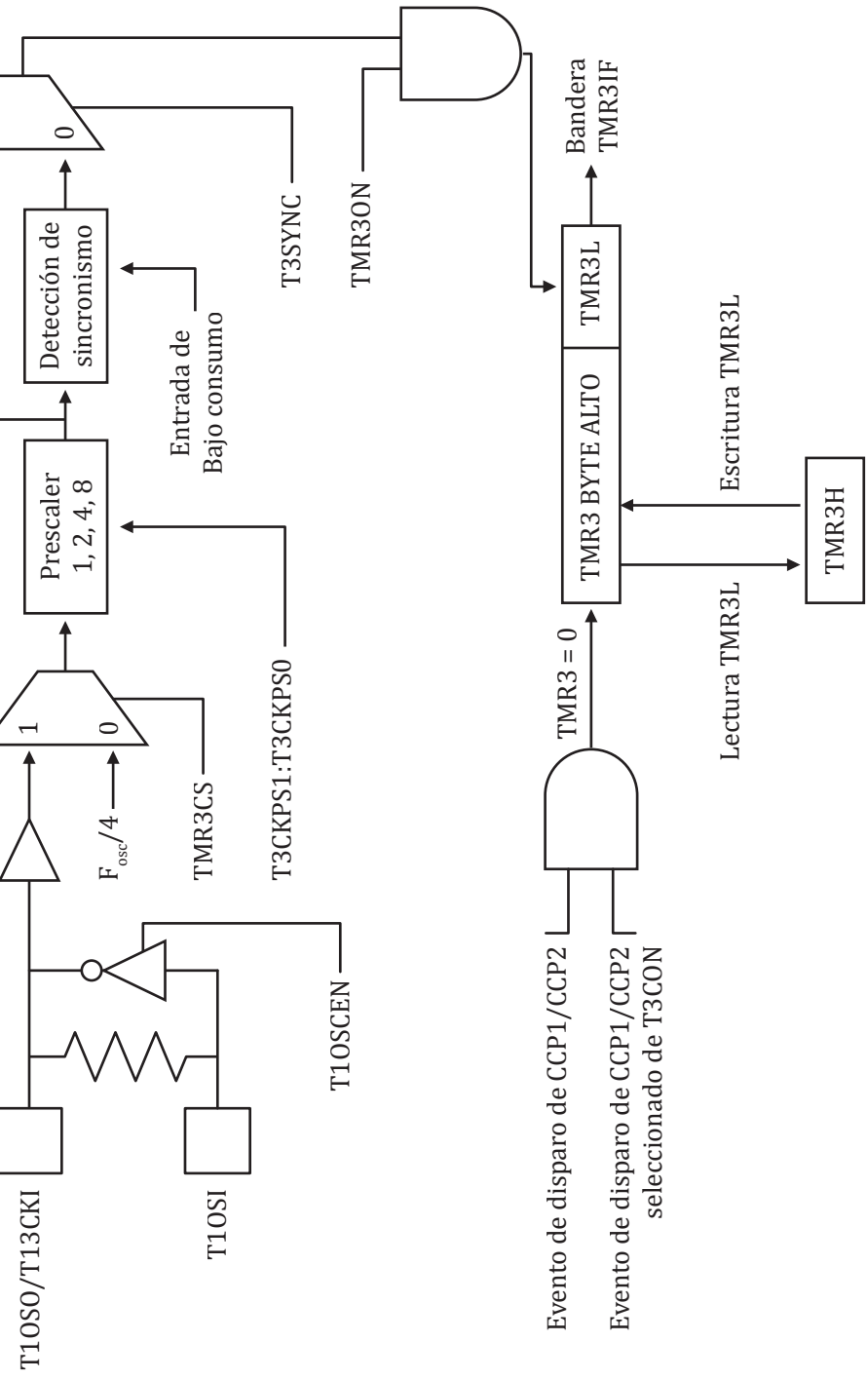
Los bits T3CCP2 y T3CCP1 permiten configurar la influencia de los eventos del módulo CCP sobre el *Timer* 3 y el *Timer* 1. En el gráfico 9.14 se observan las combinaciones binarias para cada configuración.

Los bits T3CKPS1 y T3CKPS0 permiten seleccionar el divisor del Prescaler. Estos bits configuran cuatro divisores diferentes. Su configuración se muestra en el gráfico 9.15.

El bit T3SYNC tiene funcionalidad cuando el bit TMR3CS es '1' y se ha seleccionado la fuente de reloj externa. Con el bit T3SYNC se selecciona el sincronismo de la señal de reloj externa colocando el valor a nivel '0'.

El bit TMR3CS permite seleccionar la fuente de reloj. Si la fuente es externa, este bit se pone a nivel '1'. Las fuentes de reloj externa pueden ser una señal cuadrada conectada al pin RC0 o un cristal externo de baja frecuencia. Si el bit TMR3CS se pone a nivel '0', entonces la señal de reloj de conteo es la frecuencia de instrucciones del procesador.

Gráfico 9.12. Diagrama de bloques del módulo temporizador TMR3



Elaboración propia.

Gráfico 9.13. Registro de configuración T3CON

RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON

Elaboración propia.

Gráfico 9.14. Bits de configuración de los módulos CCP en los temporizadores TMR1 y TMR3

T3CCP2	T3CCP1	Configuración
0	0	El *Timer* 1 es la fuente de reloj de comparación y captura para ambos módulos CCP.
0	1	El *Timer* 3 es usado como fuente de reloj para el módulo CCP2 y el *Timer* 1 para el módulo CCP1.
1	0	El *Timer* 3 es la fuente de reloj de comparación y captura para ambos módulos CCP.
1	1	

Elaboración propia.

Gráfico 9.15. Bits de configuración del Prescaler del módulo TMR3

T3CKPS1	T3CKPS0	Prescaler
0	0	1
0	1	2
1	0	4
1	1	8

Elaboración propia.

Finalmente, el bit TMR3ON permite habilitar la funcionalidad del módulo TMR3. Si este bit se pone a nivel '1', el módulo queda habilitado para el conteo.

El evento de interrupción del módulo TMR3 se genera cuando la bandera TMR3IF del registro PIR2 se pone a nivel '1' y el bit TMR3IE del registro PIE2 se ha habilitado. El evento de interrupción es generado por el desbordamiento del registro TMR3. Los eventos de entrada de captura solo ponen la cuenta de TMR3 en el valor 0x0000.

9.2 Diseño de un semáforo sincronizado con el *Timer* 0

Una aplicación interesante para poner a prueba el funcionamiento del temporizador *Timer* 0 es el diseño de un semáforo digital con contador de tiempo en luz verde y luz roja, similar a los sistemas de semáforos de las ciudades grandes. En el gráfico 9.16 se puede ver una imagen de uno de estos semáforos. Los semáforos contienen un segundero que realiza una cuenta regresiva e indica en dos Displays 7 segmentos de alta luminosidad y gran tamaño la cantidad de segundos que restan para que la luz cambie de estado. Normalmente, estos segunderos tienen color verde cuando la luz del semáforo es de

este color y son de luz roja cuando el semáforo se encuentra en luz de color rojo. La luz ámbar se activa cuando el contador de luz verde se encuentra en los tres últimos segundos antes de llegar a ser 00.

La propuesta de diseño de un semáforo digital con contador se muestra en el gráfico 9.17. Allí se pueden observar los siguientes detalles del diseño:

1. El controlador del semáforo es un PIC18F4550.
2. Los cuatro Displays 7 segmentos se conectan al Puerto B del microcontrolador. Cada pin de este puerto se conecta a uno de los cuatro Displays 7 segmentos tipo ánodo común. El pin RB0 se conecta con los segmentos A, el pin RB1 con los segmentos B, el pin RB2 con los segmentos C, el pin RB3 con los segmentos D, el pin RB4 con los segmentos E, el pin RB5 con los segmentos F, el pin RB6 con los segmentos G y el pin RB7 con los segmentos que cumplen la función de punto decimal.
3. El ánodo de cada Display 7 segmentos se conecta al colector de un transistor PNP modelo 2N3906. Los emisores de estos transistores se conectan a 5 V y cada una de las bases se conectan a un pin del Puerto A y E a través de una resistencia de 4.7 kΩ. El transistor opera en corte y saturación. Cuando el pin respectivo conectado a la resistencia de base genera 0 V, entonces el transistor entra en saturación, permitiendo el encendido del Display 7 segmentos que tiene conectado. Con 5 V en la resistencia de base, el transistor entra en corte, impidiendo el paso de la corriente y, por tanto, manteniendo el Display 7 segmentos apagado, independientemente de los valores de voltaje generados en cada segmento.
4. Conectados a los pines RD1, RD2 y RD3 se encuentran tres LED que cumplen con la función de encender o apagar la luz de semáforo. El pin RD1 controla a la luz verde, el pin RD2 a la luz ámbar y el pin RD3 a la luz roja.
5. Los displays D1 y D2 representan las cifras del contador para luz verde, mientras que los Displays 7 segmentos D3 y D4 representan la cifra numérica para la luz roja. El contador de luz roja empieza en 69 segundos y el contador de luz verde arrancará en 59 segundos.

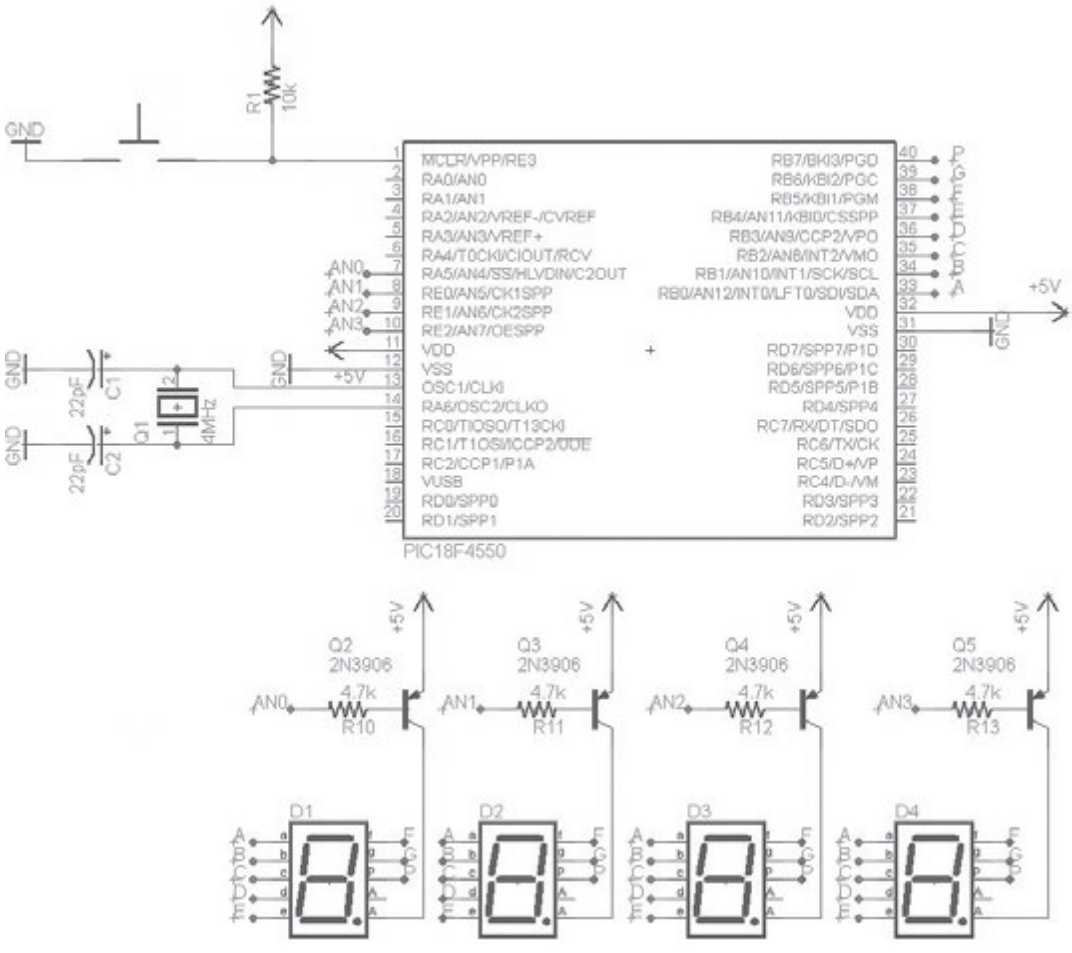
Gráfico 9.16. Semáforo digital con contador de tiempo

Fuente: www.costosperu.com/ap-site-noticias-informacion.php?seccion=&noticia=1785

El tema clave en esta aplicación es que los cuatro Displays 7 segmentos están conectados al mismo bus generador de caracteres que corresponde al Puerto B. Si se envía un código binario por este puerto, llegará a los cuatro displays simultáneamente. Es con el transistor que controla el ánodo de cada display que se seleccionará cuál estará activo y cuál no. La única forma de mostrar dígitos diferentes en cada display es colocando en corte tres transistores y dejando uno activo. En ese momento, el Puerto B generará el código binario específico para la cifra de ese display. Por ejemplo, si se quiere mostrar en los cuatro displays la secuencia numérica 1, 2, 3, 4, se debe realizar el siguiente proceso:

1. Poner en saturación el transistor Q2 y en corte los otros tres transistores. Colocar en el Puerto B el código binario 0xF9 que genera el número 1 en el Display 7 segmentos D1. Esperar un tiempo t_1.
2. Poner en saturación el transistor Q3 y en corte los otros tres. Colocar en el Puerto B el código binario 0xA4 que corresponde al número 2 en el Display 7 segmentos D2. Esperar un tiempo t_1.
3. Poner en saturación el transistor Q4 y en corte los otros tres. Colocar en el Puerto B el código binario 0xB0 que corresponde al número 3 en el Display 7 segmentos D3. Esperar un tiempo t_1.
4. Poner en saturación el transistor Q5 y en corte los otros tres. Colocar en el Puerto B el código binario 0x99 que corresponde al número 4 en el Display 7 segmentos D4. Esperar un tiempo t_1.

Gráfico 9.17. Circuito esquemático para la propuesta de diseño de un semáforo digital con contador de tiempo



Elaboración propia.

La pregunta importante es la siguiente: ¿cuánto debe ser el tiempo t_1 a fin de que la persona visualice los cuatro displays encendidos simultáneamente y sin fluctuaciones? Lo que se sabe es que el ojo humano no puede detectar una frecuencia de encendido y apagado mayor que 50 Hz. Esto significa que mientras el periodo de encendido y apagado de cada display sea menor que 1/50 Hz = 20 milisegundos, la persona visualizará de manera continua el encendido de las cifras en cada display.

Para este ejemplo, se usará el cristal de 4 MHz externo sin PLL, con lo cual cada ciclo de instrucción será de 1 µsegundo. Si se configura el Prescaler del *Timer* 0 en 16 y su modo de operación en 8 bits, entonces cada conteo del registro TMR0 se realizará en 16 µsegundos y, por tanto, el tiempo de desbordamiento se realizará en 256×16 µsegundos, que equivale a 4096 µsegundos o 4.096 milisegundos.

En cada evento de desbordamiento se realizará un cambio en la activación de los transistores. Se requieren cuatro eventos para recorrer los cuatro displays en un ciclo completo. Esto implica que el periodo de encendido y apagado de cada display es de 4096 milisegundos×4 = 16 384 milisegundos, que corresponde a una frecuencia aproximada de 61 Hz, lo cual será visualizado por el ojo humano como una luz constante y no fluctuante.

En cada uno de los eventos de desbordamiento se actualizará el valor de cada display. Para esto, lo más recomendable es hacer uso de la rutina de interrupción por desbordamiento del TMR0. Luego, en una rutina principal, se llevará el conteo descendente de las cifras a mostrar en los displays. En una tabla se almacenarán los códigos binarios que dan forma a los dígitos numéricos en el Display 7 segmentos. El punto decimal del display se mantendrá siempre apagado, con lo cual RB7 será '1' de forma constante. Conforme se lleva la cuenta de 69 a 0 en los displays D3 y D4, el LED rojo deberá estar encendido. Si la cuenta se lleva de 59 a 0 en los displays D1 y D2, el LED verde estará encendido y los otros dos apagados. Solo cuando la cuenta sea menor o igual a 3, el LED ámbar se encenderá y el LED verde tendrá que permanecer apagado. En el programa 9.2 se muestra la solución a este problema aplicando lenguaje ANSI C.

Programa 9.2. Solución del problema del semáforo en lenguaje ANSI C

```
#include <xc.h>
#pragma config FOSC   = XT_XT          //XT_XT Opción de reloj del CPU.
#pragma config PLLDIV = 1              //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV = OSC1_PLL2      //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON
#define ROJO       LATDbits.LATD1
#define AMBAR   LATDbits.LATD2
```

```
#define VERDE    LATDbits.LATD3
#define LUZ_ROJA  0
#define LUZ_VERDE 1
#define LUZ_AMBAR 2
#define HABILIT_MSB_ROJO    LATAbits.LATA5
#define HABILIT_LSB_ROJO    LATEbits.LATE0
#define HABILIT_MSB_VERDE  LATEbits.LATE1
#define HABILIT_LSB_VERDE   LATEbits.LATE2
#define DELAY1SEG Delay10KTCYx(100)
unsigned char sseg[]={0xC0,0xF9,0x24,0xB0,0x99,0x92,0x83,0xF8,0x80,0x98};
char num_rojo=69,num_verde=0xFF;
char estado = LUZ_ROJA;
void main()
{
        TRISD = 0xF0;
        TRISA = 0x0F;
        TRISE = 0xF0;
        TRISB = 0x80;
        T0CON = 0x43; //Prescaler = 1/16
        INTCONbits.TMR0IE = 1;
        INTCONbits.GIE    = 1;
        T0CONbits.TMR0ON  = 1;
        ROJO = 1;
        while(1)
        {
                if(estado == LUZ_ROJA)
                {
                        DELAY1SEG;
                        num_rojo--;
                        if(num_rojo == 0)
                        {
                                num_verde = 59;
                                num_rojo  = 0xFF;
                                estado = LUZ_VERDE;
                                VERDE = 1;
                                ROJO  = 0;
                                AMBAR = 0;
                        }
                }
                else if(estado == LUZ_AMBAR)
                {
```

```
                    DELAY1SEG;
                    num_verde--;
                    if(num_verde == 0)
                    {
                        estado = LUZ_ROJA;
                        num_rojo = 69;
                        num_verde = 0xFF;
                        AMBAR  = 0;
                        ROJO   = 1;
                    }
                }
                else if(estado == LUZ_VERDE)
                {
                    DELAY1SEG;
                    num_verde--;
                    if(num_verde == 5)
                    {
                        estado = LUZ_AMBAR;
                        VERDE = 0;
                        AMBAR = 1;
                    }
                }
        }
}
void interrupt RUT_INT(void)
{
   static char digito=0;
   if(INTCONbits.TMR0IF==1)
   {
        switch(digito)
        {
            case 0:
                HABILIT_LSB_ROJO  = 1;
                HABILIT_MSB_VERDE = 1;
                HABILIT_LSB_VERDE = 1;
                HABILIT_MSB_ROJO  = 0;
                if(num_rojo==0xFF) LATB = 0xFF;
                else          LATB = sseg[num_rojo/10];
            break;
            case 1:
                HABILIT_MSB_ROJO  = 1;
```

```
                    HABILIT_MSB_VERDE = 1;
                    HABILIT_LSB_VERDE = 1;
                    HABILIT_LSB_ROJO  = 0;
                    if(num_rojo==0xFF) LATB = 0xFF;
                    else     LATB = sseg[num_rojo%10];
                    break;
                case 2:
                    HABILIT_MSB_ROJO  = 1;
                    HABILIT_LSB_ROJO  = 1;
                    HABILIT_LSB_VERDE = 1;
                    HABILIT_MSB_VERDE = 0;
                    if(num_verde == 0xFF) LATB = 0xFF;
                    else     LATB = sseg[num_verde/10];
                break;
                case 3:
                    HABILIT_MSB_ROJO  = 1;
                    HABILIT_LSB_ROJO  = 1;
                    HABILIT_MSB_VERDE = 1;
                    HABILIT_LSB_VERDE = 0;
                    if(num_verde == 0xFF)  LATB = 0xFF;
                    else       LATB = sseg[num_verde%10];
                break;
        }
        digito++;
        if(digito==4) digito=0;
        INTCONbits.TMR0IF = 0;
   }
}
```

9.3 Diseño de un reloj en tiempo real con el *Timer* 1

Un reloj en tiempo real es aquel capaz de poder llevar la cuenta de la hora y la fecha por muchos años sin sufrir un retraso. Esto, como se sabe, es posible si se utiliza como fuente de reloj de conteo un cristal de 32 768 kHz, que tiene una frecuencia potencia de 2, con lo cual un temporizador puede hacer un conteo a esta frecuencia generando un evento de desbordamiento exactamente durante cada segundo.

Los relojes en tiempo real usados en la práctica son, por lo general, circuitos integrados con un protocolo de comunicación serial síncrono que se encargan de llevar la cuenta de la hora y fecha de manera automática. Cuentan con una entrada de voltaje de 3 V proveniente de una batería de litio que garantiza la actualización de la hora mientras el circuito esté desenergizado. Este tipo de dispositivo se verá más adelante en el capítulo 12.

Por el momento, para seguir con los ejemplos de funcionamiento de los temporizadores internos del PIC18F, se creará un reloj en tiempo real de segundos, minutos, horas, días, meses y años. Claro está que este programa perderá el conteo de la hora y la fecha si es que se retira la energía del microcontrolador. El reloj funcionará de la siguiente manera:

1. Cada segundo, la variable *seg* se incrementará en un dígito decimal. La variable *dias* se inicializa en 1 y la variable *mes* también. La variable *ano* se inicializa en 2014.
2. Si la variable *seg* es 60, inmediatamente tomará el valor de 0 y se incrementará la variable *min*.
3. Si la variable *min* es 60, inmediatamente tomará el valor de 0 y se incrementará la variable *hor*.
4. Si la variable *hor* es 24, inmediatamente tomará el valor de 0 y se incrementará la variable *dias*.
5. Si la variable *dias* toma el valor de 31 o 32, se verificará el valor de la variable *mes* para determinar si se debe incrementar. Por ejemplo, si *mes == 1* (enero), entonces *dias* deberá ser 32 para pasar al valor de 1 y *mes* tome el valor 2. Si *mes == 2* (febrero), entonces *dias* deberá ser 29 para incrementar *mes* y tomar el valor de 1.
6. Finalmente, si la variable *mes* toma el valor de 13, inmediatamente se le asigna el valor 1 y la variable *ano* se incrementará en 1.

En el programa 9.1 se vio un ejemplo de configuración del *Timer* 1 usando un cristal externo de 32 768 kHz. El temporizador se configuró para generar una interrupción cada segundo exacto. Esto se logra cargando el valor 0x80 en el registro TMR1H. Con lo cual la cuenta arrancará en 32 768 y, cuando se desborde, habrá pasado exactamente un segundo. Esta rutina se utilizará en el programa de ejemplo del reloj en tiempo real para poder generar la secuencia de incremento del segundero representado en la variable *seg*.

En el programa 9.3 se muestra la solución a este problema haciendo uso del lenguaje ANSI C. Aquí se utiliza un cristal externo de 4 MHz con PLL activo para obtener 48 MHz de frecuencia de reloj; y, por lo tanto, 12 MHz de frecuencia de instrucciones. Todo el proceso de actualización de la hora y fecha reside en la rutina de interrupción de desbordamiento del TMR1, la cual se ejecuta exactamente cada segundo transcurrido.

Programa 9.3. Programa de generación de un reloj en tiempo real

```
#include <xc.h>
#include <delays.h>

#pragma config FOSC   = XTPLL_XT       //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV = 1              //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV = OSC1_PLL2      //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
```

```
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON

char dias_mes[]={31,28,31,30,31,30,31,31,30,31,30,31};
char bandera_1seg=0;
char seg = 49, min = 59, hor = 23;
char dia = 31, mes = 12, ano = 12;
void main()
{
	LATD  = 0;
	TRISD = 0;
	TMR1H = 0x80;
	TMR1L = 0x00;
	T1CON = 0x0F; //Se selecciona el oscilador de 32,768kHz
	PIE1bits.TMR1IE = 1;
	INTCONbits.PEIE = 1;
	INTCONbits.GIE  = 1;
	while(1)
	{
	if(bandera_1seg == 1)
		{
			seg++;
			if(seg==60) {seg=0;min++;}
			if(min==60) {min=0;hor++;}
			if(hor==24) {hor=0;dia++;}
			if(dia > dias_mes[mes-1])
			{
				mes++;
				dia = 1;
			}
			if(mes==13) {mes=1;ano++;}
			bandera_1seg = 0;
		}
	}
}
void interrupt RUT_INT(void)
{
	if(PIR1bits.TMR1IF==1)
	{
```

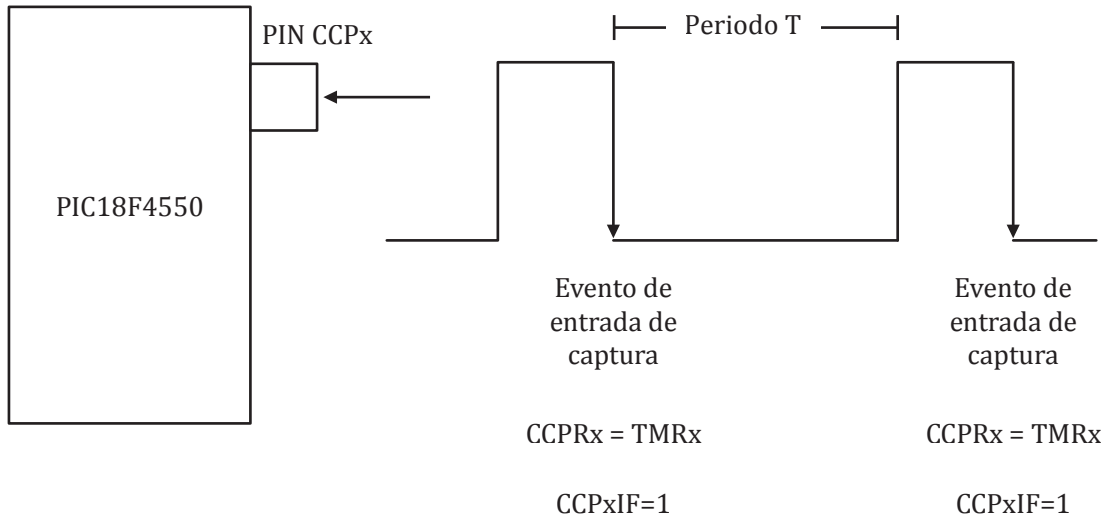
```
                TMR1H = 0x80;
                TMR1L = 0;
                bandera_1seg = 1;
                PIR1bits.TMR1IF = 0;
        }
}
```

9.4 El módulo de entrada de captura

La familia de microcontroladores PIC18F cuenta con dos módulos CCP (Captura, Comparación y PWM) que permiten obtener el periodo de una onda de entrada (Captura), generar una señal cuadrada de un periodo determinado (Comparación) y producir una señal de ancho de pulso variable (PWM). Las primeras dos operaciones están relacionadas con los *Timers* 1 y 3, mientras que la generación de PWM queda para uso exclusivo del *Timer* 2. Los módulos de captura se denominan CCP1 y CCP2.

En el diagrama del gráfico 9.18 se tiene una señal periódica, con un periodo de T. Con el módulo de entrada de captura es posible medir dicho periodo. El primer paso es configurar el módulo CCPx (donde x puede ser 1 o 2) para que opere como entrada de captura. Luego, se debe seleccionar un flanco activo, el cual generará un evento que pondrá la bandera CCPxIF (donde x puede ser 1 o 2) en nivel '1' (pudiendo activarse una interrupción en este caso). En el gráfico 9.18 se muestra que el flanco activo es de bajada.

Gráfico 9.18. Diagrama de operación del módulo de entrada de captura para el cálculo de periodos de una señal



Elaboración propia.

Cuando ocurre un evento de entrada de captura (flanco de bajada de la señal), la bandera CCPxIF se pone a nivel '1'. En ese preciso instante el registro CCPRx toma el valor del temporizador TMRx (donde x puede ser 1 o 3) manteniendo dicho valor hasta que se genere el siguiente evento. Como ejemplo, para el primer evento TMRx = 1000 y para el segundo evento TMRx = 3000. El temporizador se configuró con un Δt de 8 μsegundos. Entonces, el periodo T tendrá una duración de (3000-1000)×Δt = 16 milisegundos. De esta manera, el programa puede conocer el periodo de una señal externa y así será posible analizarla. Más adelante en el presente capítulo se verán ejemplos aplicados para este módulo.

Cabe mencionar que los temporizadores contarán libremente en el proceso de entrada de captura. Cada vez que el temporizador se desborde, el bit respectivo TMRxIF se pondrá a nivel '1'. El programador deberá borrar esta bandera cada vez que detecte el desbordamiento.

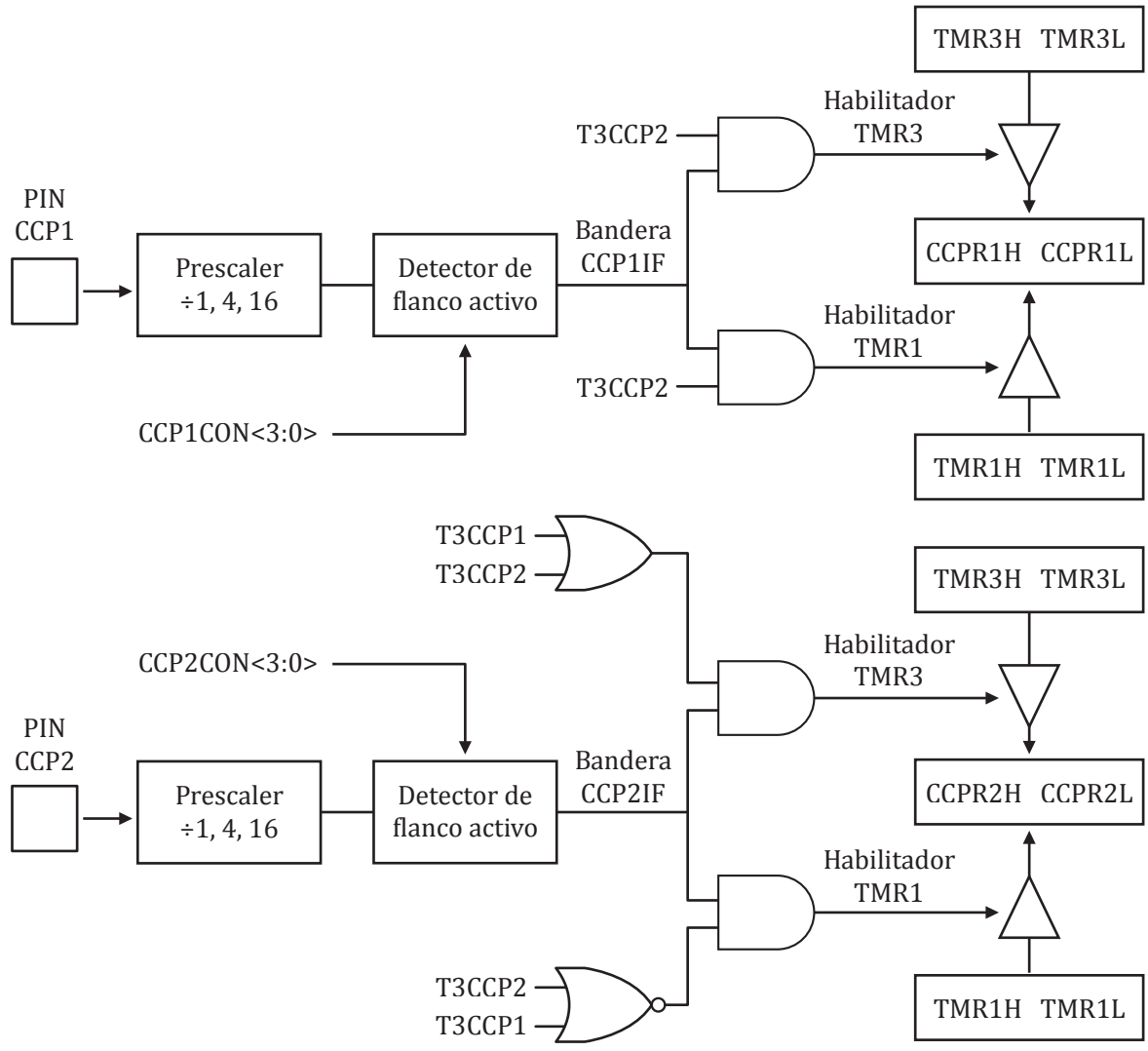
La selección del temporizador 1 o 3 para los modos de entrada de captura o comparación de salida depende de la configuración de los bits en el registro T3CON. Por lo general, un temporizador puede asociarse a ambos módulos CCP1 o CCP2 si es que están configurados en el mismo modo de operación (captura o comparación). Si cada módulo se configura en otro modo, entonces se deberá utilizar otros temporizadores.

En el microcontrolador PIC18F4550, el pin CCP1 corresponde a la línea RC2 y CCP2 corresponde al pin RC1. Ambos pines deben ser configurados como entradas digitales antes de iniciar los modos de configuración de la entrada de captura.

9.4.1 Registros de configuración

En el gráfico 9.19 se observa un diagrama de bloques de los módulos de entrada de captura del PIC18F, donde las señales de entrada ingresan a un Prescaler con divisor programable de 1, 4 o 16. Esto implica que se puede detectar el tiempo entre todos los flancos activos de la señal externa, o cada cuatro flancos activos o cada 16 flancos, respectivamente.

Gráfico 9.19. Diagrama de bloques de los módulos de entrada de captura del PIC18F



Elaboración propia.

El flanco activo solo se puede configurar en dos modos: flanco de subida o de bajada. En el caso de que el Prescaler se haya configurado en 4 o 16, el flanco activo por defecto es el de subida. Una vez que se detecta el flanco activo, la bandera CCPxIF se pone a nivel '1', generando que el contenido del registro TMR3 o TMR1 se cargue en el registro CCPR1x. La elección del temporizador de carga en CCPR1x depende de los bits T3CCP2 y T3CCP1 del registro T3CON.

El registro CCP1CON permite configurar el módulo CCP1, mientras que el registro CCP2CON configura el módulo CCP2. Ambos registros son idénticos en su función, así que en el gráfico 9.20 se muestran de forma general los bits de ambos registros y sus respectivas funciones.

Gráfico 9.20. Registro CCPxCON (donde x puede ser 1 o 2)

--	--	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0

Elaboración propia.

Las funciones de los bits de los registros CCPxCON son las siguientes:

Bits DCxB1 y DCxB0: representan los bits menos significativos del registro CCPR1L, con lo cual forman una cifra de 8 bits usada para la configuración del ancho de pulso en el módulo PWM. En los modos de entrada de captura y comparación de salida no tienen uso.

Bits CCPxM3:CCPxM0: estos bits permiten configurar el modo de operación del módulo CCPx. En el gráfico 9.21 se observan las combinaciones de estos bits y las configuraciones que producen.

Gráfico 9.21. Bits de configuración CCPxM3:CCPxM0 de los registros CCPxCON

CCPxM3	CCPxM2	CCPxM1	CCPxM0	Configuración
0	0	0	0	CCPx deshabilitado.
0	0	0	1	Reservado.
0	0	1	0	Modo de comparación con inversión de bit al evento CCPxIF = 1.
0	0	1	1	Reservado.
0	1	0	0	Modo de captura: para todo flanco de bajada.
0	1	0	1	Modo de captura: para todo flanco de subida.
0	1	1	0	Modo de captura: para todo cuarto flanco de subida.
0	1	1	1	Modo de captura: para todo dieciseisavo flanco de subida.
1	0	0	0	Modo de comparación, con el pin CCPx en bajo y luego en alto cuando CCPxIF = 1.
1	0	0	1	Modo de comparación, con el pin CCPx en alto y luego en bajo cuando CCPxIF = 1.
1	0	1	0	Modo de comparación que genera una interrupción de software cuando CCPxIF = 1.
1	0	1	1	Modo de comparación que genera eventos especiales como Reset del temporizador, conversión A/D cuando CCPxIF = 1.
1	1	0	0	Modo PWM.
1	1	0	1	
1	1	1	0	
1	1	1	1	

Elaboración propia.

9.4.2 Ejemplo de detección de teclas pulsadas de un control remoto de televisión marca Sony

Un control remoto infrarrojo transmite códigos binarios que representan la ubicación de la tecla pulsada. Estos controles abundan en el mercado nacional y presentan un LED infrarrojo en la parte superior. Dicho LED emite un haz de luz infrarroja que se encuentra modulado a una frecuencia estándar de 38 kHz. Esta frecuencia evita que otras fuentes de radiación infrarroja interfieran sobre la señal enviada por el control remoto. En el gráfico 9.22 se muestra una imagen de un control remoto estándar.

Gráfico 9.22. Control remoto estándar para televisión

Fuente: http://www.ehowenespanol.com/conectar-control-remoto-televisor-como_8702/

Para la recepción de la señal de luz se puede utilizar un receptor infrarrojo comercial similar al que se muestra en el gráfico 9.23. Este receptor se alimenta con una tensión de 5 V. Genera a su salida un nivel de tensión '0' durante la detección de luz infrarroja modulada a 38 kHz y 5 V cuando no ha detectado señal alguna.

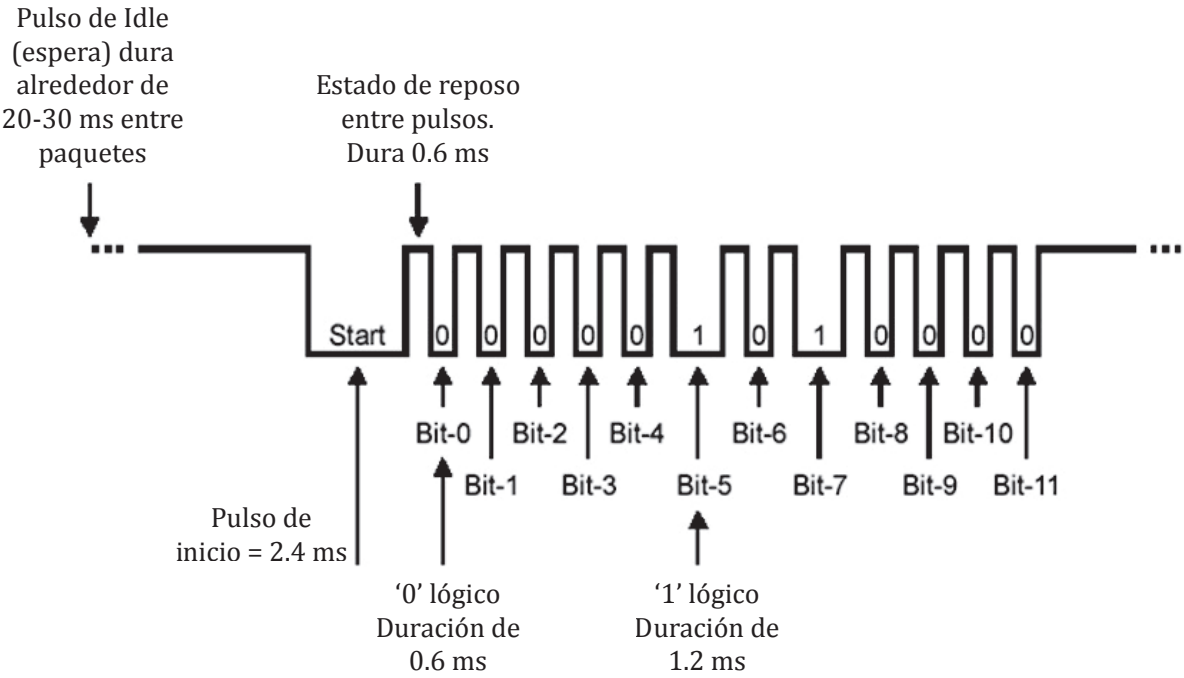
Los controles remotos de televisión se caracterizan por usar su propio protocolo de comunicación. Es decir, cada señal de datos a ser modulada presenta su propio tiempo de bit y formato de codificación binaria. La gran ventaja de los controles remotos de televisión es que sus formatos de transmisión han permanecido invariantes en el tiempo, lo cual permite que un control remoto de un televisor antiguo basado en la tecnología de tubo de rayos catódicos permita controlar un televisor actual de tecnología LED de la misma marca.

Gráfico 9.23. Receptor infrarrojo para control remoto de TV

Fuente: http://www.sigmaelectronica.net/gp1um28yk-p-134.html

El control remoto para televisión marca Sony presenta una trama binaria codificada en 12 bits de longitud. Al ser excitado el receptor infrarrojo por la señal del control remoto, se generará una trama cuyo formato se muestra en el gráfico 9.24. En esta trama se pueden observar los 12 bits que componen la información. El pulso de *Start* o Inicio sirve para que el controlador del receptor pueda sincronizarse y detectar fácilmente el punto donde comienza la trama de datos. Seguido de este pulso continúan los bits de datos. El primer bit se considera el LSB y son los 7 primeros bits de la trama los que representan el código de la tecla pulsada. En este ejemplo, el código es 0100000_2. Luego, los últimos 5 bits siempre toman el valor 1000_2, el cual es un código de verificación.

Gráfico 9.24. Trama de datos recibida generada por la pulsación de una tecla en un control remoto de televisor marca Sony



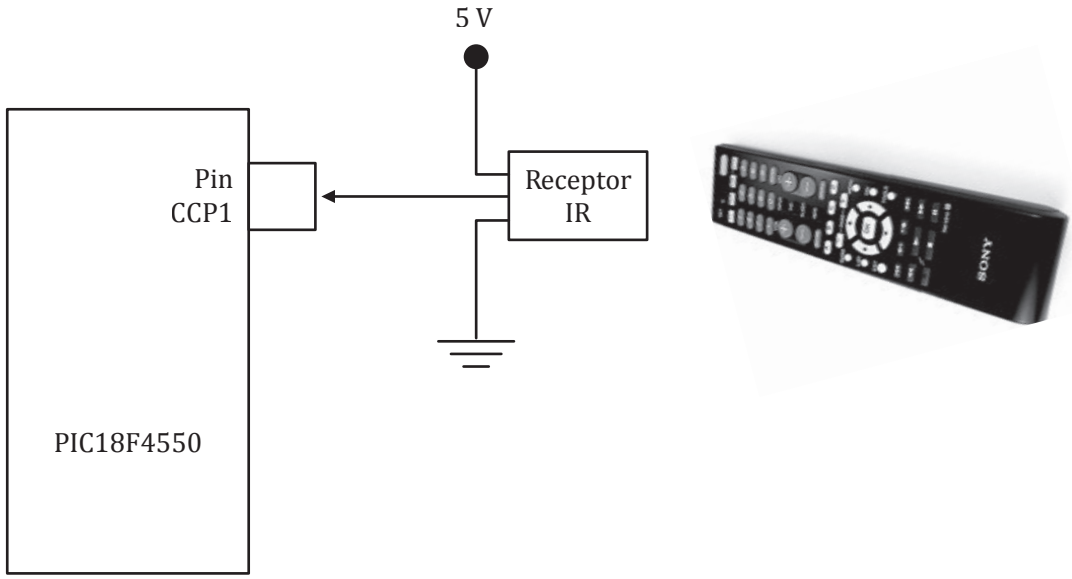
Elaboración propia.

En el gráfico 9.25 se muestra la propuesta de solución para el diseño de un sistema detector de teclas pulsadas de un control remoto de televisor marca Sony. La salida del receptor infrarrojo se conecta al pin CCP1, la cual será analizada mediante el modo de entrada de captura sensible a flancos de subida. De esta manera, al detectarse el primer flanco de subida, se estará ante el final del pulso de inicio. Luego de este instante aparecerán doce flancos adicionales correspondientes a los 12 bits o periodos de la señal.

Cada periodo de la señal recibida por el receptor infrarrojo puede tener dos valores: 1.2 milisegundos ('0') y 1.8 milisegundos ('1'). Entonces, a partir del segundo evento de captura se podrá obtener el valor de la cuenta del temporizador entre el flanco actual y el flanco anterior. Conociendo el tiempo se puede determinar si ese periodo corresponde a un nivel alto o bajo de la trama de datos. En el programa 9.4 se muestra la solución a este problema.

Como se puede observar en el programa, se configura la frecuencia de instrucciones a 12 MHz y el Prescaler del *Timer* 1 (asociado a la Entrada de captura en el pin CCP1), a 8. Entonces, el tiempo de conteo del TMR1 será $\frac{8}{12MHz} = 0.667\,\mu seg$. De esta manera, para un tiempo transcurrido de 1.2 milisegundos el TMR1 realizará un conteo aproximado de $\frac{1200\,\mu seg}{0.667\,\mu seg} \approx 1799$, y para un tiempo transcurrido de 1.8 milisegundos este mismo temporizador realizará un conteo aproximado de $\frac{1800\,\mu seg}{0.667\,\mu seg} \approx 2699$.

Gráfico 9.25. Propuesta de solución para un sistema detector de teclas pulsadas haciendo uso de un control remoto marca Sony para televisor



Elaboración propia.

Programa 9.4. Algoritmo de detección del código enviado por un control remoto para televisor marca Sony

```
#include <xc.h>

#pragma config FOSC   = XTPLL_XT      //XT_PLL_XT Opción de reloj del CPU
#pragma config PLLDIV = 1             //Divisor del PLL (4MHz/1)*24=96MHz
#pragma config CPUDIV = OSC1_PLL2     //Divisor del Postscaler 96MHz/2 = 48MHz
#pragma config PWRT   = ON            //Power Up Timer
#pragma config WDT    = OFF           //Watch Dog Timer
#pragma config LVP    = OFF           //Low Voltage ICSP
#pragma config BOR    = OFF           //Brown Out Reset
#pragma config DEBUG  = ON            //Background Debugger Enable
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF           //Port B pins configured as AD
#pragma config MCLRE  = ON            //Reset pin configuration
```

```
typedef union{
        struct{
                unsigned char bit0: 1;
                unsigned char bit1: 1;
                unsigned char bit2: 1;
                unsigned char bit3: 1;
                unsigned char bit4: 1;
                unsigned char bit5: 1;
                unsigned char bit6: 1;
                unsigned char bit7: 1;
        }_Bits;
        unsigned char dato_TV;
}SONY;
SONY dato_rx, cod_error;
unsigned int val_cap,aux;

char bandera = 0;

void main()
{
        TRISD = 0x00;
        TRISC = 0xBF;
        TRISB = 0;
        LATD = 0;
        T1CONbits.RD16   = 1;
        T1CONbits.T1CKPS1 = 1;
        T1CONbits.T1CKPS0 = 1;          //Prescaler = 8
        T1CONbits.TMR1ON  = 1;
        CCP2CON = 0x05;                 //Modo captura todo flanco de subida
        PIE2bits.CCP2IE   = 1;
        PIE1bits.TMR1IE   = 1;
        INTCONbits.PEIE   = 1;
        INTCONbits.GIE    = 1;
        while(1)
        {
                if(bandera==1)
                {
                if(cod_error.dato_TV == 0x01)
                   {
                    //Se detectó correctamente la tecla pulsada
                        LATB = dato_tx.dato_TV;
                    }
```

```
                    else
                    {
                      //Error, escriba su código de error aquí:
                    }
                      dato_rx.dato_TV = 0;
                      cod_error.dato_TV = 0;
                      bandera = 0;
                }
        }
}

void interrupt RUT_INT(void)
{
        static unsigned char vez = 0;
        if(PIR1bits.TMR1IF==1)
        {
                PIR1bits.TMR1IF=0;
        }
        if(PIR2bits.CCP2IF ==1)
        {
                val_cap = CCPR2;
                TMR1L = 0;
                TMR1H = 0;
                switch(vez)
                {
                        case 1:
                                if(val_cap>=2600 && val_cap<2800)
                    dato_rx._Bits.bit0 = 1;
                                else if(val_cap>= 1700 && val_cap<1900)
                    dato_rx._Bits.bit0 = 0;
                                else
                    cod_error._Bits.bit7 = 1;
                        break;
                        case 2:
                                if(val_cap>=2600 && val_cap<2800)
                    dato_rx._Bits.bit1 = 1;
                                else if(val_cap>= 1700 && val_cap<1900)
                    dato_rx._Bits.bit1 = 0;
                                else
                    cod_error._Bits.bit7 = 1;
                        break;
                        case 3:
```

```
                if(val_cap>=2600 && val_cap<2800)
dato_rx._Bits.bit2 = 1;
                else if(val_cap>= 1700 && val_cap<1900)
dato_rx._Bits.bit2 = 0;
                else
cod_error._Bits.bit7 = 1;
        break;
        case 4:
                if(val_cap>=2600 && val_cap<2800)
dato_rx._Bits.bit3 = 1;
                else if(val_cap>= 1700 && val_cap<1900)
dato_rx._Bits.bit3 = 0;
                else
cod_error._Bits.bit7 = 1;
        break;
        case 5:
                if(val_cap>=2600 && val_cap<2800)
dato_rx._Bits.bit4 = 1;
                else if(val_cap>= 1700 && val_cap<1900)
dato_rx._Bits.bit4 = 0;
                else
cod_error._Bits.bit7 = 1;
        break;
        case 6:
                if(val_cap>=2600 && val_cap<2800)
dato_rx._Bits.bit5 = 1;
                else if(val_cap>= 1700 && val_cap<1900)
dato_rx._Bits.bit5 = 0;
                else
cod_error._Bits.bit7 = 1;
        break;
        case 7:
                if(val_cap>=2600 && val_cap<2800)
dato_rx._Bits.bit6 = 1;
                else if(val_cap>= 1700 && val_cap<1900)
dato_rx._Bits.bit6 = 0;
                else
cod_error._Bits.bit7 = 1;
        break;
        case 8:
                if(val_cap>=2600 && val_cap<2800)
cod_error._Bits.bit0 = 1;
```

```
                    else if(val_cap>= 1700 && val_cap<1900)
    cod_error._Bits.bit7 = 1;
            break;
            case 11:
                    if(val_cap>=2600 && val_cap<2800)
    cod_error._Bits.bit3 = 1;
                    else if(val_cap>= 1700 && val_cap<1900)
    cod_error._Bits.bit3 = 0;
                    else
    cod_error._Bits.bit7 = 1;
            break;
            case 12:
                    if(val_cap>=2600 && val_cap<2800)
    cod_error._Bits.bit4 = 1;
                    else if(val_cap>= 1700 && val_cap<1900)
    cod_error._Bits.bit4 = 0;
                    else
    cod_error._Bits.bit7 = 1;
            break;
            default:
            break;
}
vez++;
if(vez == 13)
    cod_error._Bits.bit0 = 0;
                    else
    cod_error._Bits.bit7 = 1;
            break;
            case 9:
                    if(val_cap>=2600 && val_cap<2800)
    cod_error._Bits.bit1 = 1;
                    else if(val_cap>= 1700 && val_cap<1900)
    cod_error._Bits.bit1 = 0;
                    else
    cod_error._Bits.bit7 = 1;
            break;
            case 10:
                    if(val_cap>=2600 && val_cap<2800)
    cod_error._Bits.bit2 = 1;
                    else if(val_cap>= 1700 && val_cap<1900)
    cod_error._Bits.bit2 = 0;
                    else
{
```

```
                    bandera = 1;
                    vez = 0;
                    Delay10KTCYx(240);
                }

                PIR2bits.CCP2IF=0;
        }
        if(PIR1bits.RCIF==1)    PIR1bits.RCIF=0;
}
```

Si se analiza el gráfico 9.24, una vez detectado el primer flanco de subida precederán 12 flancos adicionales. El tiempo transcurrido entre cada dos flancos consecutivos presenta la codificación del bit. Si en cada evento de interrupción de entrada de captura por flanco de subida el registro TMR1 se inicializa en 0x0000, en el siguiente evento o interrupción se tendrá en el TMR1 la cuenta completa empezando desde un valor conocido (0x0000). Esto permitirá rápidamente evaluar el tiempo transcurrido: si el TMR1 es 2699, habrán transcurrido 1.8 milisegundos y la codificación corresponderá a un '1'. Si el TMR1 es 1799, habrán transcurrido 1.2 milisegundos y la codificación corresponderá a un '0'. Sin embargo, en el programa se establecen rangos de tiempo para cada periodo, ya que una serie de factores pueden afectar la calidad de la señal y el periodo detectado no necesariamente será de 1.2 o 1.8 milisegundos exactos. Estos factores pueden ser la distancia del control remoto al receptor infrarrojo, la carga de las baterías del control remoto, la calidad del control remoto (puede ser un control remoto universal que no genera una trama exactamente idéntica a la original), entre otros. Los rangos establecidos para el conteo del TMR1, entonces, corresponden a 1700 y 1900 (1.13 y 1.26 milisegundos) para el periodo de 1.2 milisegundos y de 2600 a 2800 (1.73 y 1.86 milisegundos) para el periodo de 1.8 milisegundos. Cualquier valor de conteo detectado fuera de este rango en uno de los periodos de la señal puede significar dos cosas: mala calidad de la señal recibida o una trama correspondiente a otra marca de control remoto. Ante esta situación se genera una condición de error y no se considera la recuperación del código binario de la trama recibida.

En el programa 9.4 no se analiza la trama recibida, simplemente se ubican los bits en el orden recibido dentro de un tipo de variable que combina una unión con una estructura de 8 campos de 1 bit y se obtiene un código binario de 7 bits. Para las teclas numéricas este código corresponde a cifras de 0x00 a 0x09, donde la tecla 1 es 0x00 y la tecla 0 es 0x09. Para la pulsación de otras teclas, como CH+, CH-, VOL+ o VOL-, aparecerán otros códigos binarios. Se recomienda al lector completar este programa para asociar el código binario recibido por el símbolo asociado en cada una de las teclas pulsadas del control remoto.

9.5. El modo de comparación de salida

El modo de comparación de salida de los módulos CCP1 y CCP2 opera de una manera muy diferente al modo de entrada de captura. En primer lugar, en este modo la señal cuadrada es generada por el microcontrolador. Los registros de configuración cumplen con la función de determinar el periodo de la señal a generar y seleccionar la transición de los niveles en los pines de salida digital CCP1 y CCP2.

Los temporizadores TMR1 y TMR3 son utilizados para el modo de comparación. Aquí se selecciona un temporizador para que comience a contar libremente cada cierto tiempo Δt configurado por el factor del Prescaler del registro TxCON (donde x puede ser 1 o 3, dependiendo del uso del *Timer* 1 o 3, respectivamente). El registro CCPRx ahora se carga con un valor de 16 bits. Cuando la cuenta del temporizador iguala al valor almacenado en CCPRx un comparador genera un evento, haciendo que la bandera CCPxIF sea '1'. Producto de este evento se pueden producir los siguientes casos:

1. El pin CCPx se pone a nivel '1'.
2. El pin CCPx se pone a nivel '0'.
3. El pin CCPx cambia de nivel (de '1' a '0' o de '0' a '1').
4. No varía su estado.

Por ejemplo, si el TMR1 se pone a contar libremente con un Δt de 8 μsegundos y el registro CCPR1 se carga con el valor 10 000. Entonces, cada 8 μsegundos × 10 000 = 80 milisegundos se generará un evento de comparación, donde el estado del pin CCP1 podrá variar.

9.5.1 Registros de configuración

El diagrama de bloques del modo de comparación de salida se muestra en el gráfico 9.26. En este esquema se observa que tanto el valor de los temporizadores TMR1 como TMR3 pueden ser usados para compararse tanto con el registro CCPR1 como con CCPR2. Cuando los valores son iguales, las banderas CCP1IF y CCP2IF toman el valor de '1', lo cual puede disparar una interrupción si es que los bits CCP1IE y CCP2IE son '1'. El bloque de lógica de salida permite configurar la acción que generará el evento de comparación, controlando el estado del pin CCP1 o CCP2 y también generando un evento especial, como puede ser una conversión analógica digital o un Reset del TMR3. Como se vio en el subcapítulo anterior, el modo de operación de la lógica de salida depende de los registros CCP1CON y CCP2CON. Finalmente, cabe mencionar que es necesario que los pines RC2 y RC1 se configuren como salidas digitales para poder operar en este modo.

En el programa 9.5 se muestra un ejemplo de configuración del modo de comparación de salida. En él se configura la lógica de salida del módulo CCP1 para que el pin RC2 cambie de estado automáticamente en cada comparación. El pin RD0 se pone a nivel '0' durante cada comparación de salida, y a '1' en cada desbordamiento del TMR1. El cristal usado para este programa es de 4 MHz sin uso del PLL interno, por lo cual la frecuencia de instrucciones es de 1 MHz.

Programa 9.5. Ejemplo de generación de un pulso cuadrado con el modo de comparación de salida

```
        org 0x00
        goto MAIN
        org 0x08
        goto RUT_INT
        org  0x18
        retfie
        org 0x20
MAIN:
        clrf  TRISC
        clrf  TRISD
        bcf   T1CON,T1OSCEN
        bcf   T1CON,TMR1CS          ;Selección de la frecuencia de instrucciones.
        bcf   T1CON,T1CKPS0
        bcf   T1CON,T1CKPS1         ;Prescaler en 1.
        bcf   T3CON,T3CCP1          ;El Timer 1 es la fuente de reloj para
        bcf   T3CON,T3CCP2          ;CCP1 y CCP2.
        clrf  TMR1L
        clrf  TMR1H
        bsf   T1CON,TMR1ON          ;Timer 1 ON.
        movlw b'00000010'           ;Comparación de salida con el
        movwf CCP1CON               ;pin CCP1 invirtiéndose por comparación.
        movlw 0x10
        movwf CCPR1L
        movlw 0x27
        movwf CCPR1H                ;CCPR1 = 10,000.
        bsf   PIE1, CCP1IE          ;Se activa la interrupción por comparación.
        bsf   PIE1, TMR1IE          ;Se activa la interrupción por desbordamiento
        bsf   INTCON,PEIE           ;del TMR1.
```

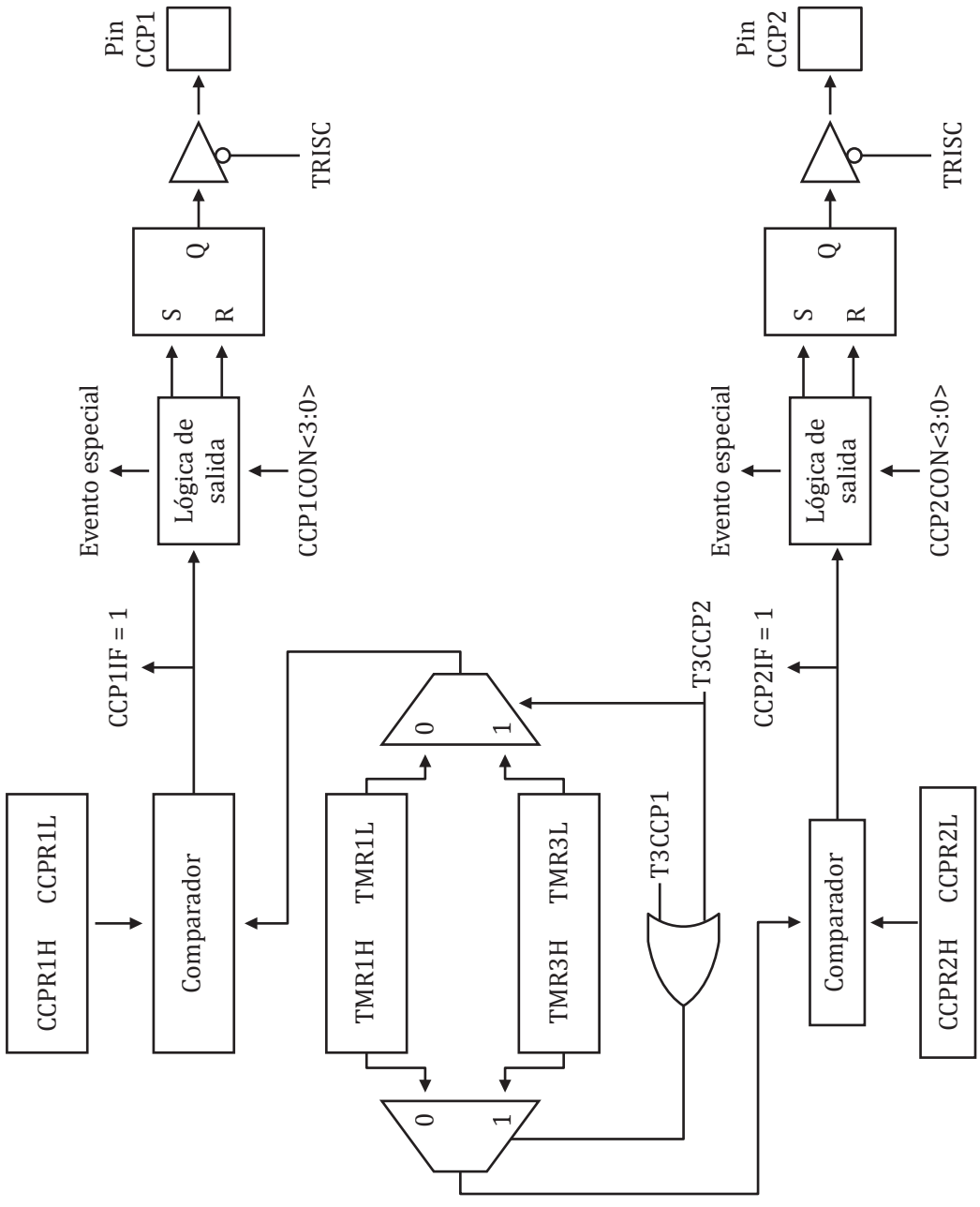
```
        bsf   INTCON,GIE
LAZO:
        goto  LAZO
RUT_INT:
        btfsc PIR1,TMR1IF
        goto  RUT_TMR1
        btfsc PIR1,CCP1IF
        goto  RUT_CCP1
        retfie
RUT_CCP1:
        bcf   LATD,0
        bcf   PIR1,CCP1IF
        retfie
RUT_TMR1:
        bsf   LATD,0
        bcf   PIR1,TMR1IF
        retfie
        END
```

En el gráfico 9.27 se muestran los resultados del programa 9.5. Primero se analiza la señal en RC2. Este pin cambia de estado durante cada evento de comparación que ocurre en un tiempo T_3, cada vez que TMR1 = 10 000. Pues bien, el periodo de conteo del TMR1 se ha seleccionado en el programa con un Prescaler de 1, por lo tanto la cuenta presenta un Δt = 1 μsegundo.

Dado que el cambio de RC2 ocurre cada vez que el TMR1 toma el valor de 10 000 y, obviamente, este proceso se genera cuando el TMR1 ha contado de 10 000 hasta 65 535 y luego de 0 a 10 000, esto implica que el TMR1 ha recorrido 65 536 cuentas entre cada comparación. Por lo tanto, el tiempo T_3 es el siguiente:

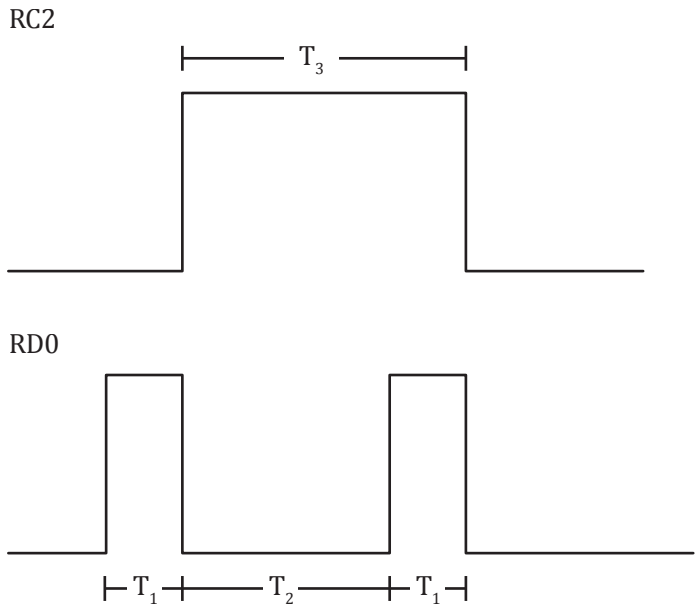
$$T_3 = 65536 \times 1 \mu segundos = 65{,}536 \mu segundos = 65.536 \ milisegundos$$

Gráfico 9.26. Diagrama de bloques del modo de comparación de salida



Elaboración propia.

Gráfico 9.27. Resultado de la ejecución del programa 9.5



Elaboración propia.

Por otro lado, la señal generada por el pin RD0 toma el valor de '1' cada vez que ocurre un desbordamiento de TMR1, y toma el valor de '0' en un evento de comparación. La duración de T_1 representa el tiempo transcurrido entre el desbordamiento de TMR1 y el evento de comparación. Este tiempo es el siguiente:

$$T_2 = 65.536\ milisegundos\ -\ 10\ milisegundos\ = 55.536\ milisegundos$$

Finalmente, el tiempo T_1 corresponde a la diferencia entre el periodo de desbordamiento y el tiempo de comparación, y se obtiene de la siguiente manera:

$$T_1 = 65.536\ milisegundos\ -\ 55.536\ milisegundos = 10\ milisegundos$$

9.5.2 Ejemplo de control de posición de un servomotor

Un servomotor es un dispositivo pequeño por lo general, que cuenta con un eje controlado que permite girar un engranaje en un determinado ángulo. Este puede ser llevado a posiciones específicas con precisión de 1° sexagesimal al enviar una señal codificada en PWM. Dicha señal debe mantenerse constante en la entrada del servomotor para que la posición angular también se mantenga constante. Cuando la señal codificada cambia, la posición angular de los piñones también varía. En la práctica, los servomotores son utilizados para posicionar palancas, timones, en robótica, control de posicionamiento de cámaras de vigilancia, entre otros. En el gráfico 9.28 se observa una imagen típica de un servomotor.

Gráfico 9.28. Imagen de un servomotor típico para aplicaciones de robótica

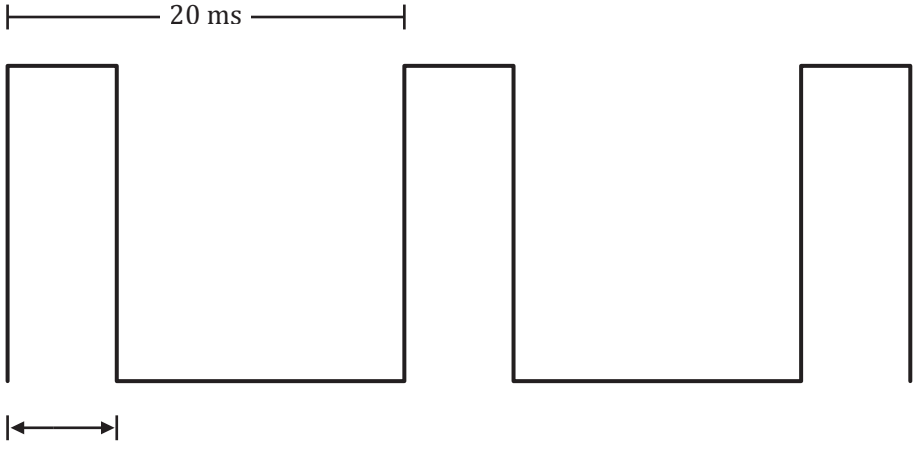
Fuente: http://electrosome.com/interfacing-servo-motor-with-8051-using-keil-c/

El motor del servomotor posee una circuitería de control y un potenciómetro. Este último se encuentra conectado al eje central del servomotor. Este potenciómetro permite a la electrónica de control supervisar el ángulo actual del servomotor. Si el eje está en el ángulo adecuado, entonces el motor estará apagado; si el circuito detecta que el ángulo no es el correcto, el motor girará en la dirección adecuada hasta llegar al ángulo correcto. El eje del servomotor es capaz de girar alrededor de unos 180° sexagesimales. En algunos modelos de servomotor este ángulo puede llegar a los 210°. Un servomotor, en condiciones normales, se utiliza para controlar un movimiento angular de entre 0° y 180°.

Los servomotores presentan tres líneas de conexión: alimentación, tierra y la entrada para la señal de control. Se controlan mediante el envío de un pulso PWM, el cual debe ser generado por un controlador externo. Los parámetros a controlar de esta señal son el mínimo pulso y el máximo pulso, que deben ser generados de manera repetitiva. De no ser así, el servomotor colocará su eje en una posición neutral.

El ángulo de giro se determina por la duración del pulso que se aplica a la señal de control. El periodo de la señal de control es por lo general de 20 milisegundos. El ancho de pulso debe variar alrededor de 1.5 milisegundos, entre 0.5 milisegundos y 2.5 milisegundos, tal como se muestra en el gráfico 9.29, para lograr generar el ángulo de movimiento entre 0° y 180°. Un valor de 1.5 milisegundos hará que el eje permanezca en la posición neutral.

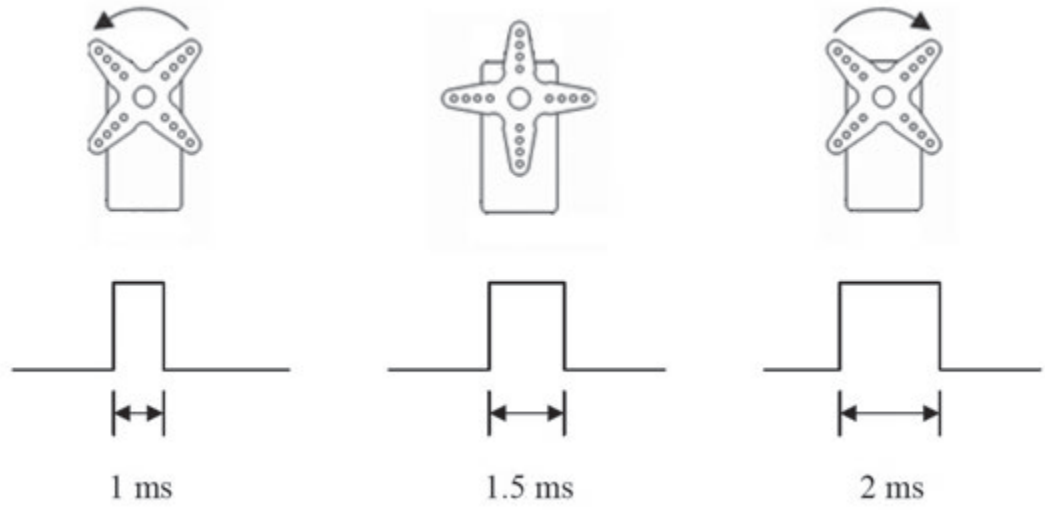
Gráfico 9.29. Formato de la señal de control de posición del servomotor



Elaboración propia.

El ancho de pulso debe generarse de forma repetitiva para que el servomotor mantenga su posición. Cuando el pulso presenta una duración de 1.5 milisegundos, mantiene su posición neutral. Si el ancho de pulso aumenta por encima de este valor, el eje del servomotor se desplaza en sentido horario hasta 90° sexagesimales de la posición por defecto. Si el ancho de pulso es menor, entonces el desplazamiento es en sentido antihorario, hasta unos 90° sexagesimales de la posición por defecto, tal como se muestra en el gráfico 9.30.

Gráfico 9.30. Desplazamiento del eje del servomotor según el ancho de pulso de la señal de control



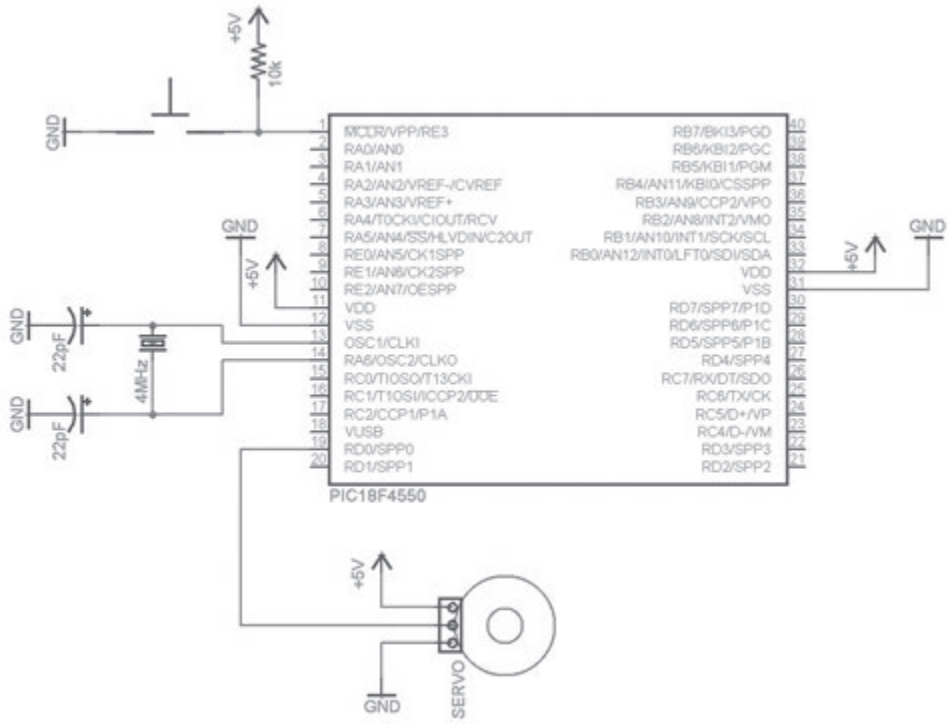
Elaboración propia.

En el gráfico 9.31 se muestra la conexión propuesta del PIC18F4550 al servomotor. Como se puede observar, la línea RD0 genera la señal de PWM para variar cada segundo la posición de giro del servomotor entre los ángulos +90° y -90°. Para esto, se utiliza una frecuencia de reloj de 4 MHz y se fija el Prescaler del *Timer* 1 en 1, de manera que se obtiene un conteo cada 1 µsegundo. En el programa 9.6 aparece el código demostrativo.

De esta manera, el programa genera dos anchos de pulso: de 0.5 milisegundos y de 2.5 milisegundos, ambos dentro de un periodo de onda de 20 milisegundos requerido por el servomotor. Para el primer ancho de pulso, el eje del motor girará -90°; y para el segundo, +90°. Cada segundo se alterará el valor de comparación del *Timer* 1 a fin de que se pueda actualizar el nuevo ancho de pulso. Por cada periodo aparecen dos eventos de comparación: para generar el pulso de nivel alto y para generar el de nivel bajo de la señal. Si el primer evento de comparación lo produce un lapso de 0.5 milisegundos, el segundo evento se producirá luego de 19.5 milisegundos; mientras que si el primer evento de comparación lo produce un lapso de 2.5 milisegundos, el segundo evento se producirá luego de 17.5 milisegundos. Este proceso se puede observar en la interrupción por comparación.

Para esta aplicación lo ideal hubiera sido utilizar el módulo PWM del PIC18F4550. Sin embargo, como se verá en el siguiente subcapítulo, este módulo depende del temporizador TMR2 de 8 bits y no es capaz de alcanzar un periodo de 20 milisegundos. Normalmente, este módulo opera para frecuencias mayores, con lo cual su uso en servomotores queda restringido, al menos para este modelo de microcontrolador.

Gráfico 9.31. Conexión del servomotor al PIC18F4550



Elaboración propia.

Programa 9.6. Control del movimiento del servomotor a 90° y -90° cada segundo

```
#include <xc.h>
#include <delays.h>
#pragma config FOSC   = XT_XT
#pragma config PLLDIV = 1
#pragma config CPUDIV = OSC1_PLL2
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON

#define CNTRLSERVO LATDbits.LATD0
#define _90_G    2500
#define _M90_G   500
unsigned int angulo=_90_G;
void main()
{
```

```
        TRISC =  0;
        TRISD = 0xF0;
        T1CONbits.T1OSCEN = 0;
        T1CONbits.TMR1CS  = 0;
        T1CONbits.T1CKPS0 = 0;
        T1CONbits.T1CKPS1 = 0;
        T3CONbits.T3CCP1  = 0;
        T3CONbits.T3CCP2  = 0;
        TMR1L = 0;
        TMR1H = 0;
        T1CONbits.TMR1ON  = 1;
        CCP1CON = 0b00000010;
        CCPR1 = 2000;                   //2mseg de comparación
        PIE1bits.CCP1IE = 1;
        PIE1bits.TMR1IE = 1;
        INTCONbits.PEIE = 1;
        INTCONbits.GIE  = 1;
        CNTRLSERVO     = 1;
        while(1)
        {
                Delay10KTCYx(100);     //Retardo 1 segundo
                angulo = _M90_G;
                Delay10KTCYx(100);     //Retardo 1 segundo
                angulo = _90_G;
        }
}
void interrupt RUT_INT(void)
{
        if(PIR1bits.CCP1IF==1)
        {
                TMR1L = 0;
                TMR1H = 0;
                CNTRLSERVO = ~CNTRLSERVO;
                if(CNTRLSERVO==1) CCPR1 = angulo;
                else          CCPR1 = 20000-angulo;
                PIR1bits.CCP1IF=0;
        }
        if(PIR1bits.TMR1IF==1)
        {

                PIR1bits.TMR1IF=0;
        }
}
```

9.6 El modo de modulación por ancho de pulso (PWM)

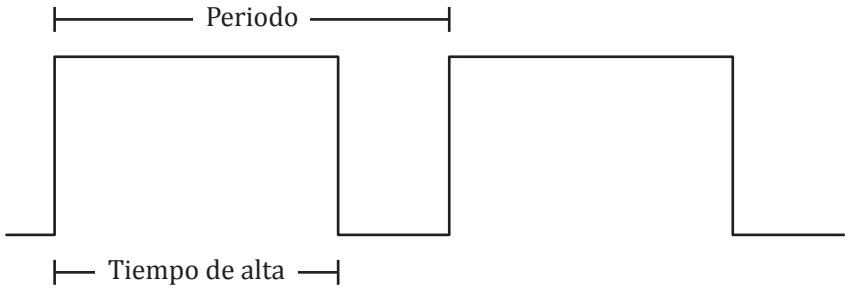
El PWM es muy utilizado para el control de velocidad de motores DC, diseño de inversores, transmisión de datos, conversión digital analógica, posicionamiento de servomotores y muchas otras aplicaciones más. Es por esta razón que es muy raro encontrar un sistema embebido que no cuente con un módulo interno para generación de señales PWM.

Una señal PWM solo existe para formas de onda cuadradas, donde la señal solo presenta dos niveles: alto y bajo. Un periodo completo de la señal se encuentra definido por el tiempo en alto y el tiempo en baja, tal como se muestra en el gráfico 9.32. El Ciclo de trabajo se define como la relación porcentual entre el tiempo en alta sobre el periodo, como se muestra en la siguiente ecuación:

$$Ciclo\ de\ trabajo = \frac{Tiempo\ en\ alta}{Periodo} \times 100\%$$

De esta manera, una señal cuyo ciclo de trabajo sea mayor al 50% tendrá un tiempo en alta más largo que el tiempo en baja y, por ende, un nivel de energía mayor. El ciclo de trabajo puede variar desde 0% (ausencia completa de la señal) hasta 100% (señal completamente en nivel '1').

Gráfico 9.32. Ejemplo de señal PWM con ciclo de trabajo mayor al 50%

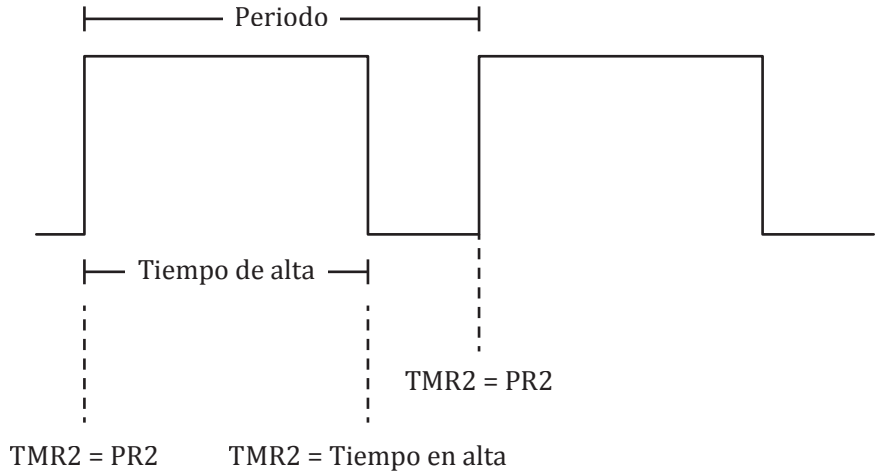


Elaboración propia.

9.6.1 Registros de configuración

La forma para generar la onda PWM en el PIC18F4550 es como se muestra en el gráfico 9.33. El temporizador TMR2 debe configurarse para que realice un conteo automático empezando por el valor 0x00 hasta el valor contenido en el registro PR2. Posteriormente, el registro CCPxRL debe cargarse con un valor numérico junto con los bits 4 y 5 del registro CCPxCON, formando una cifra de 10 bits que determina el tiempo en alta.

Gráfico 9.33. Configuración de la señal de ancho de pulso variable con el modo PWM



Elaboración propia.

La fórmula para definir el periodo de la señal del gráfico 9.33 es la siguiente:

$$Periodo = (PR2 + 1) \times 4 \times T_{OSC} \times (Prescaler\ TMR2)$$

Como se vio anteriormente, el Prescaler del TMR2 puede tomar solo tres valores: 1, 4 y 16, mientras que T_{OSC} representa el periodo de la señal de reloj del cristal externo. Para definir el tiempo en alta de la señal se utiliza la siguiente ecuación:

$$Tiempo\ en\ alta = (CCPRxL : CCPxCON < 5:4 >) \times (\mathrm{Pr}escaler\ TMR2)$$

El tiempo en alta depende de un valor de 10 bits que resulta de la concatenación de los 8 bits del registro CCPRxL con los bits 5 y 4 del registro CCPxCON. Estos últimos representan los bits menos significativos de la cifra. La idea es que el tiempo en alta siempre sea un valor menor al periodo. De no ser así, la señal de salida estará en nivel alto de forma continua.

En el gráfico 9.34 se observa el diagrama de bloques del modo PWM. Los valores de los registros CCPRxL y CCPxCON se juntan para formar una cifra de 10 bits. El valor de CCPRxL se carga en CCPRxH que cumple la función de registro Latch. Este valor se compara con los 8 bits de TMR2. Los dos bits restantes en CCPxCON<5:4> se comparan con los 2 bits internos del reloj del Prescaler de TMR2, lo cual forma una comparación neta de 10 bits. Esta comparación genera un Reset en el flip flop de salida, originando un nivel '0' en el PIN CCPx. Cuando este evento ocurre, la bandera CCPxIF se pone a nivel '1'.

Cuando se produce una comparación entre PR2 y TMR2, la bandera TMR2IF se pone a nivel '1', lo cual provoca un pulso sobre la entrada Set en el Flip Flop, generando un nivel '1' en el pin CCPx. Este evento determina que la bandera TMR2IF se ponga a nivel '1'.

9.6.2 Ejemplo de generación de una señal PWM de 40% de Duty Cycle

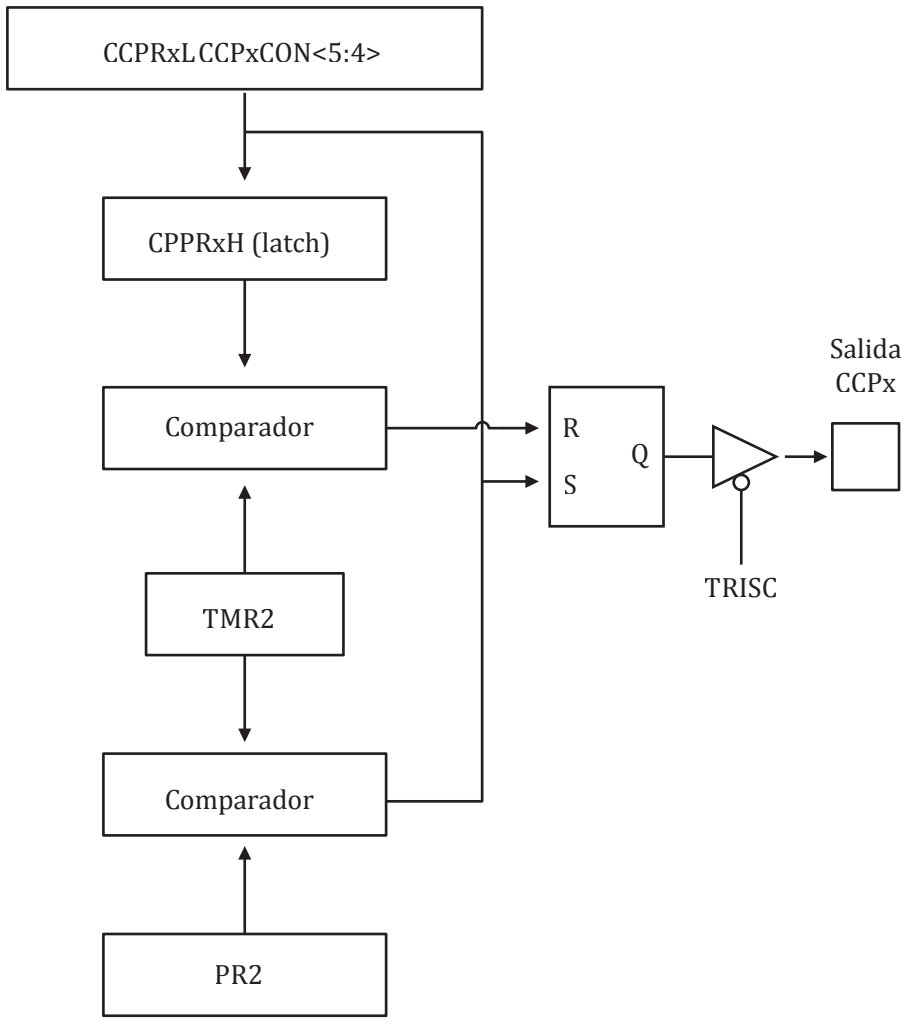
En el programa 9.7 se muestra un ejemplo de generación de una señal de PWM con 40% de ciclo de trabajo sobre el pin RC2 o CCP1. Se utiliza un cristal externo de 4 MHz sin PLL, con lo cual T_{OSC} = 0.25 µsegundos. El registro PR2 se carga con el valor 249 y el Prescaler se fija en 4, quedando el siguiente periodo:

$$Periodo = (249+1) \times 4 \times 0.25 \mu seg \times 4 = 1 \ milisegundo$$

Por otra parte, para fijar el tiempo en alta de la señal para un ciclo de trabajo del 40%, se deberá ajustar el valor de los registros CCPR1L y CCP1CON<5:4> para que la ecuación de tiempo en alta sea 400 µsegundos. Esto se logra de la siguiente manera:

$$Tiempo\ en\ alta = (400) \times 0.25 \mu segundos \times 4 = 400 \mu segundos$$

Gráfico 9.34. Diagrama de bloques del modo de generación de PWM del módulo CCPx



Elaboración propia.

Con el tiempo en alta en este valor, el ciclo de trabajo es del 40%. El siguiente paso consiste en cargar el valor 400 en los registros CCPR1L y CCP1CON<5:4>. En hexadecimal el valor se representa como 0x190. Los dos bits menos significativos son 00_2. Por lo tanto, los bits <5:4> de CCP1CON se deben mantener en nivel '0'. Los 8 bits más significativos se convierten en el código hexadecimal 0x64, el cual se carga en el registro CCPR1L.

Programa 9.7. Ejemplo de generación de una señal de PWM de 40% de ciclo de trabajo

```
        org 0x00
        goto MAIN
        org 0x08
        goto RUT_INT
        org 0x18
        retfie
        org 0x20
MAIN:
        clrf  TRISC
        bcf   PORTC,2;
        movlw .249
        movwf PR2               ;Periodo = (249+1)*4*Tosc*Prescaler.
        bcf   T2CON,T2CKPS1
        bsf   T2CON,T2CKPS0     ;Prescaler del TMR2 a 4.
        movlw 0x0E
        movwf CCP1CON           ;Modo PWM. Bits <5:4> = 00₂
        movlw 0x64
        movwf CCPR1L            ;Cargando 0x190 (400) para el Tiempo en alta
        bsf   T2CON,TMR2ON      ;TMR2 ON.
        bsf   PIE1,TMR2IE
        bsf   PIE1,CCP1IE       ;Habilitando la interrupción por comparación.
        bsf   INTCON,PEIE
        bsf   INTCON,GIE
LAZO:
        goto  LAZO
RUT_INT:
        btfsc PIR1,TMR2IF
        goto  TIMER2
        btfss PIR1,CCP1IF
        retfie
        bcf   PIR1,CCP1IF
TIMER2:
        bcf   PIR1,TMR2IF
        return
        END
```

Cabe resaltar que es importante eliminar las banderas del evento de comparación cada vez que ocurra una interrupción, para evitar que la interrupción se invoque nuevamente. Esto se realiza en la rutina RUT_INT del programa.

Video

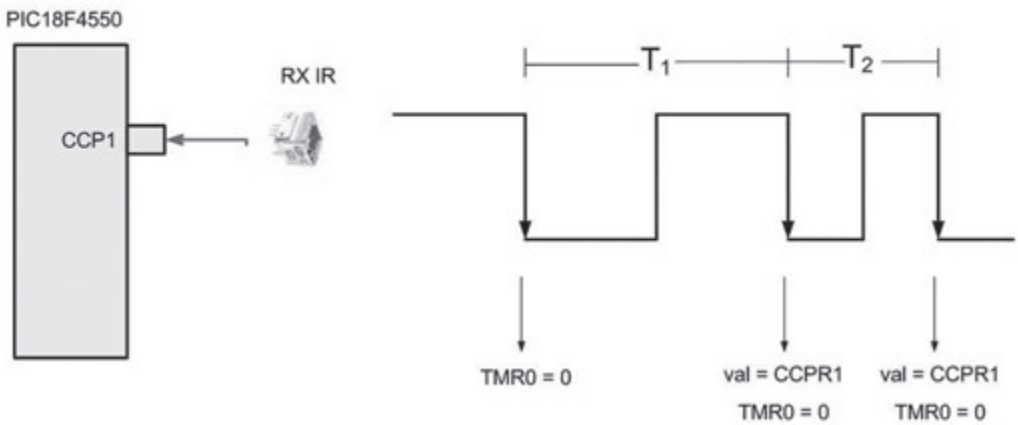
Diseño de un reloj en tiempo real utilizando el temporizador Timer 1 y el cristal externo de baja frecuencia.

http://bit.ly/1Lfp2j8

Resolución de videos en HD

Preguntas de repaso

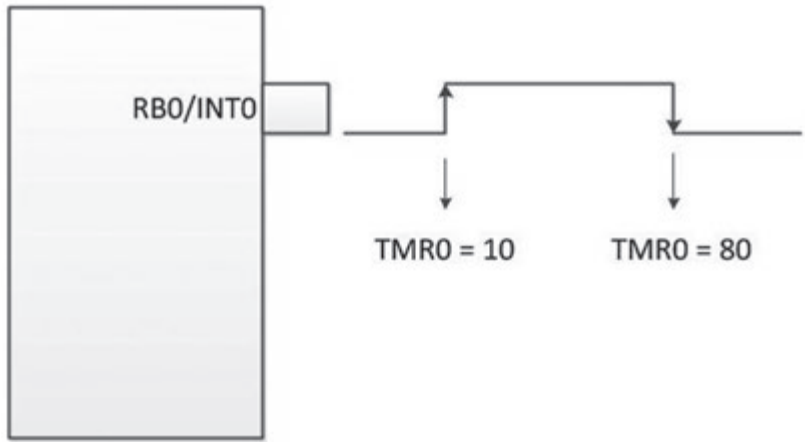
1. Un sensor infrarrojo recibe una señal de un control remoto universal de televisión que le envía dos periodos de calibración: T_1 y T_2, tal como se muestra en la figura inferior. Usted conecta la señal del receptor infrarrojo al pin CCP1 del PIC18F4550 y configura el módulo Input Capture para que opere detectando flancos de bajada sincronizando el TMR1 configurado con un Prescaler de 1 y una frecuencia de reloj del cristal externo de 4 MHz.



   Durante la llegada del primer flanco de bajada, el TMR1 se pone a 0x0000. Luego, en el segundo flanco de bajada, se obtiene el valor de CCPR1 que se carga en la variable val, la cual toma el valor de 5000. Luego, se carga el TMR1 en 0x0000 nuevamente y en el tercer flanco de bajada se obtiene un valor en CCPR1 de 2000, el cual se guarda nuevamente en la variable val.
   ¿Cuánto valen los periodos T_1 y T_2 en milisegundos?

2. Indique dos diferencias entre el *Timer* 0 y el *Timer* 1 del microcontrolador PIC18F4550.

3. En la figura inferior se observa un pulso cuadrado conectado al pin de interrupción externa RB0/INT0. Se ha diseñado un programa que tiene como objetivo hallar el tiempo de duración del pulso en nivel alto. Para esto, se habilita la interrupción externa 0, de manera que sea sensible al flanco de subida y luego al de bajada. El programa sincroniza los eventos de interrupción con el *Timer* 0. De esta forma, en la primera interrupción se detectó el valor de 10 y en la segunda el valor de 80 en el registro TMR0.



Se sabe que el PIC18F4550 se encuentra conectado a un cristal externo de 4 MHz, El *Timer* 0 utiliza como reloj de entrada la señal generada por el cristal y su Prescaler se ha configurado en 256. Determine el tiempo de duración del pulso.

4. Si una señal cuadrada periódica presenta un tiempo en alta de 10 µsegundos y su Duty Cycle es del 30% ¿Cuánto es su periodo?
5. Configure el registro CCP1CON y CCPR1L para obtener una señal cuadrada de 500 µsegundos de periodo y un Duty Cycle del 20% con un cristal externo de 4 MHz y un Prescaler de 16.

Respuestas

Respuestas del capítulo 9:

http://bit.ly/1iavFN0

Capítulo 10. El convertidor analógico digital

El PIC18F4550, al igual que la gran mayoría de microcontroladores de la familia PIC18 de Microchip, cuenta con un módulo convertidor analógico digital de 10 bits de precisión que permite la entrada de señales analógicas proveniente de sensores, lo cual es la clave para que el microcontrolador se convierta en un sistema de monitoreo o control de variables físicas. Esto permite abrir el abanico de posibilidades para la inclusión del PIC18F4550 en una gran cantidad de aplicaciones, como en industria, domótica, ingeniería biomédica, electrónica automotriz, etc.

El módulo convertidor analógico digital presenta cierto nivel de complejidad que es importante conocer, para que pueda ser configurado y utilizado de manera adecuada. Además, es necesario tener en claro conceptos sobre el proceso de digitalización de señales analógicas para poder determinar la correcta configuración del módulo en las aplicaciones con sensores.

En este capítulo se abordan los conceptos básicos de digitalización de señales analógicas, el teorema del muestreo y el funcionamiento del módulo ADC del microcontrolador. Finalmente, se presentan algunas aplicaciones prácticas que permitirán poner a prueba los conceptos explicados.

10.1 Fundamentos de un sistema de adquisición de datos

Un sistema de adquisición de datos es aquel que permite procesar una señal analógica y reproducir su forma de onda equivalente en un computador. Este sistema se utiliza en una gran cantidad de aplicaciones en las que se requiere interactuar con señales analógicas. Por ejemplo, en sistemas de control de temperatura, sistemas de monitoreo de señales biomédicas, sistemas de procesamiento digital de audio, etc.

En el gráfico 10.1 se muestra un ejemplo de un sistema de gestión de información de señales cardiacas. En algún lugar dentro de un centro médico se tiene un paciente conectado a un sistema de adquisición de datos en forma de electrocardiógrafo. El sistema de adquisición permite digitalizar las doce derivaciones cardiacas del paciente y transmitirlas a un computador vía USB o conexión Bluetooth. Este computador conectado a internet a través de un Módem-*Router* puede enlazar la información tanto con una base de datos hospitalaria como con un punto remoto donde un médico especialista puede ser capaz analizarla.

Este sistema de gestión de información es de gran utilidad para un centro médico. La parte fundamental de este sistema radica en el equipo que realiza la adquisición de señales, es decir, el electrocardiógrafo, el cual en su interior contiene un sistema de adquisición de señales que podría estar constituido por un PIC18F.

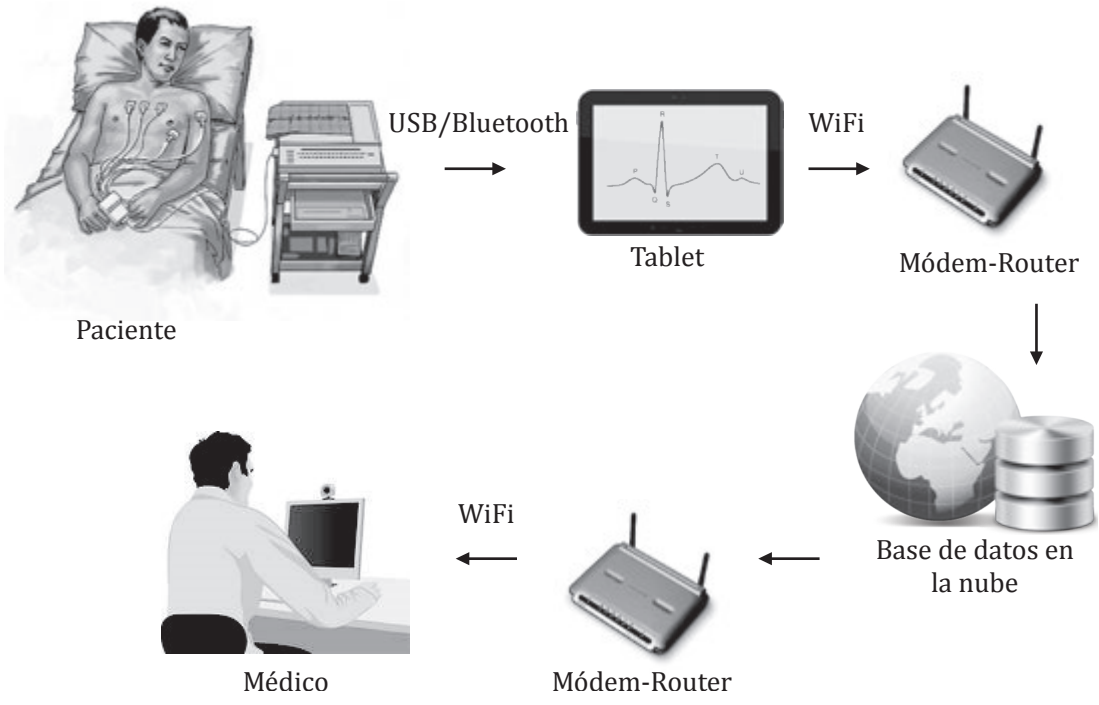
Un sistema de adquisición de señales se compone de los siguientes elementos:

1. La señal analógica producto del análisis.
2. El sensor o transductor que convertirá dicha señal a una variable física de voltaje.

3. El preamplificador de señal.
4. Filtros analógicos.
5. Etapa de amplificación y variación de nivel DC.
6. El convertidor analógico digital.
7. El controlador que se encarga de procesar las muestras digitales de la señal.

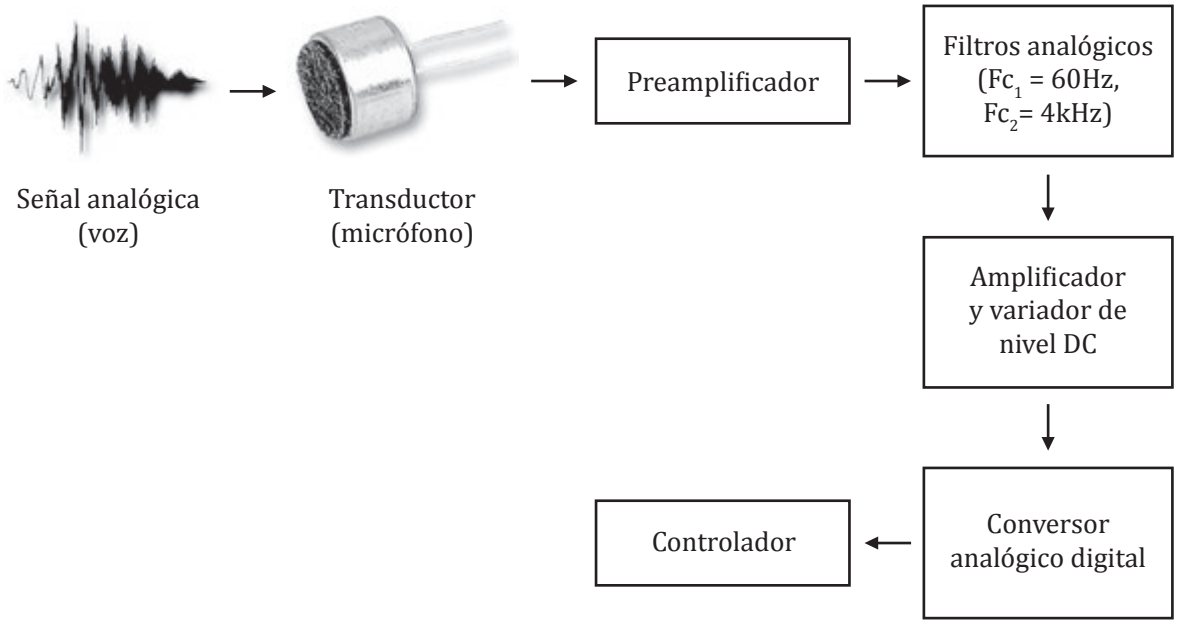
En el gráfico 10.2 se muestra la interacción de todos estos elementos en un sistema de adquisición de datos. Algunas etapas pueden variar de acuerdo con la naturaleza de la señal y las características del sensor. De igual modo, el uso de los filtros depende de estos parámetros sumados al tipo de interferencia electromagnética a la cual se encuentra sometido el circuito de adquisición.

Gráfico 10.1. Diagrama de bloques de un sistema de gestión de información de señales cardiacas



Elaboración propia.

Gráfico 10.2. Diagrama de bloques de un sistema de adquisición de datos



Elaboración propia.

En el gráfico 10.2 se observa un diagrama de bloques de un sistema de adquisición de audio. La parte más importante la conforma la señal de voz analógica que debe ser adquirida por el sistema. El transductor (micrófono, en este ejemplo) cumple con la función de transformar el dominio de la magnitud de la señal en presión de aire a niveles de voltaje equivalentes. La señal de voltaje generada por el transductor se encuentra en el orden de los milivoltios, por lo cual es necesario preamplificarla para posteriormente ajustar su amplitud. La preamplificación es un proceso que busca dar una cierta magnitud a la señal de salida del transductor lo suficientemente grande como para poder filtrar la señal, pero lo suficientemente baja como para que no se genere saturación.

Los filtros analógicos buscan acortar las bandas de frecuencia de la señal que no se necesita para el análisis. Además, tratan de eliminar aquellas frecuencias de interferencia electromagnética que se pueden montar sobre la señal de entrada. Por ejemplo, una de las frecuencias de interferencia que más afecta los sistemas de adquisición de datos son los 60 Hz de tensión de alimentación de línea. Por esta razón, en el diseño del filtro analógico se propone un filtrado pasa banda cuyas frecuencias de corte sean 60 Hz y 4 kHz, eliminando así las frecuencias de 60 Hz e inferiores y las mayores que 4 kHz (ya que la señal de voz no presenta componentes superiores a los 4 kHz). El filtrado pasa bajas tiene como fin *recortar* el ancho de banda de la señal de entrada, evitando así problemas de *Aliasing*[46]. Esto se conoce como Filtro Anti Aliasing.

El amplificador y variador de nivel DC cumple con dar a la señal la amplitud y niveles máximo y mínimo para que se adecúen al rango de entrada del conversor analógico digital. El rango de entrada debe respetarse, a fin de evitar la saturación del convertidor. El convertidor posee una resolución en

46 *Aliasing* es un efecto que ocurre en la digitalización de señales cuando se elige una frecuencia de muestreo menor al valor debido. Este efecto genera distorsión de la información.

bits fija. Finalmente, un controlador recibe las muestras digitales y las procesa para obtener la información necesaria. El controlador determina la frecuencia de muestreo de la señal.

La familia de microcontroladores PIC18F cuenta con un convertidor analógico digital que permite la digitalización de señales. Este convertidor contribuye a que el PIC18F pueda ser usado para diversas aplicaciones de control y monitoreo de variables físicas. El PIC18F, entonces, puede cumplir con la función de convertidor analógico digital y controlador del diagrama de bloques del sistema de adquisición de datos del gráfico 10.2.

A continuación se explicarán una serie de aspectos teóricos relacionados con el proceso de conversión analógica digital, que permitirán definir los criterios óptimos para realizar una correcta digitalización de una señal analógica, evitando la pérdida de información más relevante.

10.2 La conversión analógica digital

El proceso de conversión analógica digital se lleva a cabo dentro del convertidor analógico digital. Este procedimiento se divide en cuatro partes: muestreo, retención, cuantización y codificación binaria. El objetivo de este procedimiento es transformar un voltaje analógico en un código binario que represente la magnitud de dicho voltaje.

Todo convertidor analógico digital presenta limitaciones tanto en velocidad (el tiempo que tarda en convertir una muestra analógica en un código binario equivalente a su magnitud) como en resolución (número de bits del código binario generado). Además, los convertidores presentan un rango máximo de voltaje de entrada delimitado por las tensiones $VREF_L$ y $VREF_H$. Estas tres características son las más relevantes al momento de verificar si un convertidor analógico digital será el ideal para una aplicación específica.

10.2.1 Muestreo

El muestreo es el primero paso del proceso de conversión analógica digital de una señal. Este proceso consiste en multiplicar la señal analógica por un tren de impulsos unitarios separados por un tiempo Tc. Como resultado, se obtienen puntos específicos de la señal cada tiempo Tc, tal como se observa en el gráfico 10.3.

Tc se conoce como el periodo de muestreo de la señal y su valor especifica la cantidad de muestras que se pueden recolectar en un segundo, lo cual se conoce como Frecuencia de muestreo. Este último parámetro presenta unidades de Hercios (Hz) y se relaciona con el periodo de muestreo de la siguiente manera:

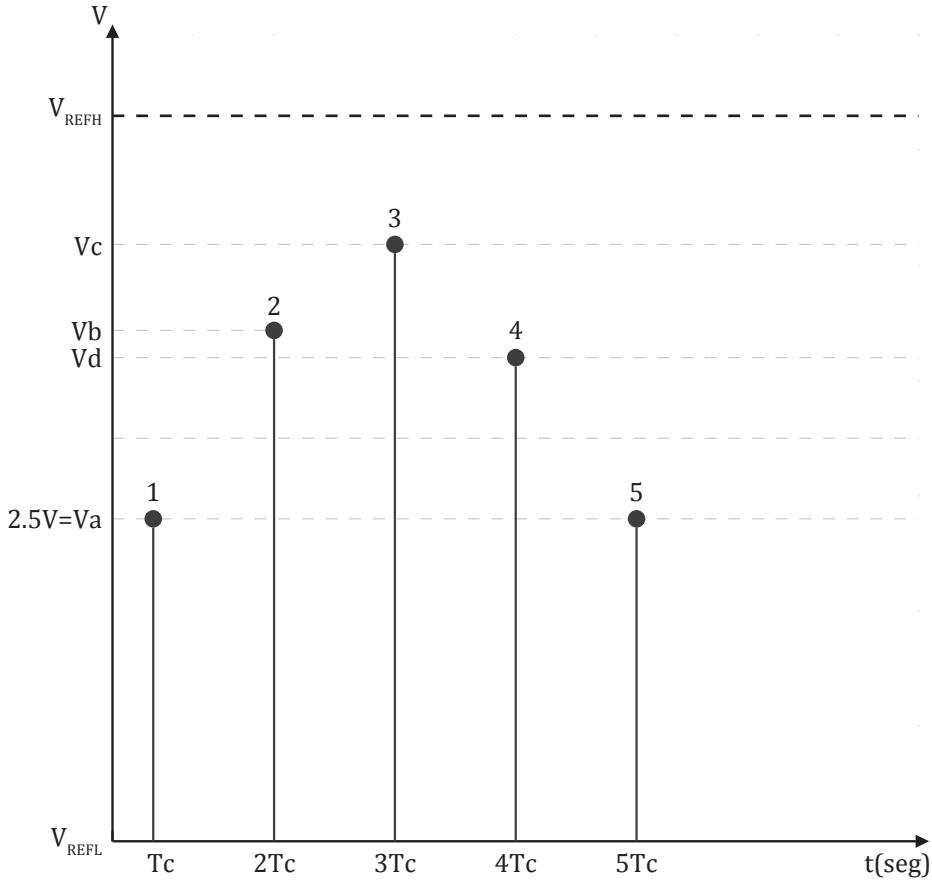
$$f_{muestreo} = \frac{1}{T_c}$$

El periodo de muestreo mínimo está limitado por el tiempo de conversión del ADC. Normalmente, la tasa de muestreo puede ser variable o configurable, pero existe un mínimo de tiempo de conversión para cada convertidor analógico digital.

Cabe resaltar que el proceso de muestreo es la primera transformación irreversible de la señal analógica, ya que requiere de una discretización de la señal en el tiempo y, por lo tanto, se pierden

todos los puntos que conforman la onda en cada intervalo XTc-(X+1)Tc, donde X = 1,2,3……N (siendo N el número total de muestras).

Gráfico 10.3. Proceso de muestreo de una señal analógica



Elaboración propia.

10.2.2 Retención

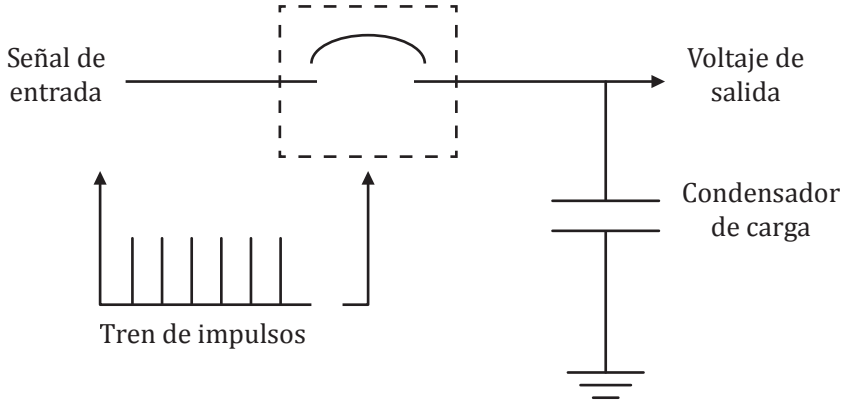
Si se observa exactamente el proceso del gráfico 10.3, se puede notar que la conversión se aplica sobre cada punto de la señal distanciado un tiempo Tc. Entonces, ¿qué ocurre durante el intervalo XTc-(X+1) Tc? Pues en este intervalo el voltaje adquirido debe ser convertido a un código binario. Para lograr esto, el valor de amplitud de la muestra tomada debe almacenarse de alguna manera para que pueda estar retenido durante el intervalo en mención. Este proceso se conoce como *Retención*.

En el diagrama del gráfico 10.4, la señal de entrada se conecta a un interruptor controlado por un tren de impulsos. Cada impulso tiene una duración muy corta y presenta un periodo Tc. El condensador se carga con el nivel de tensión de entrada y mantiene su carga durante el intervalo en el cual el interruptor se encontrará abierto. De esta manera, la tensión de la muestra tomada no se perderá y podrá ser convertida a un código binario cuando el interruptor se vuelva a abrir.

El tiempo de carga del condensador depende de su capacitancia y de la resistencia de entrada que genera el interruptor. Como se sabe, el parámetro τ equivalente a R×C, donde R es la resistencia del inte-

rruptor y C la capacitancia del condensador. τ determina el tiempo de carga y descarga del capacitor. Para que la carga sea rápida, la capacitancia y la resistencia deben ser mínimas. Claro está que esto también limita el tiempo de descarga, que puede ser menor al intervalo de muestreo Tc, lo que alteraría todo el proceso. La elección de estos valores es crítica y es una de las limitantes del periodo de muestreo.

Gráfico 10.4. Circuito de muestreo y retención



Elaboración propia.

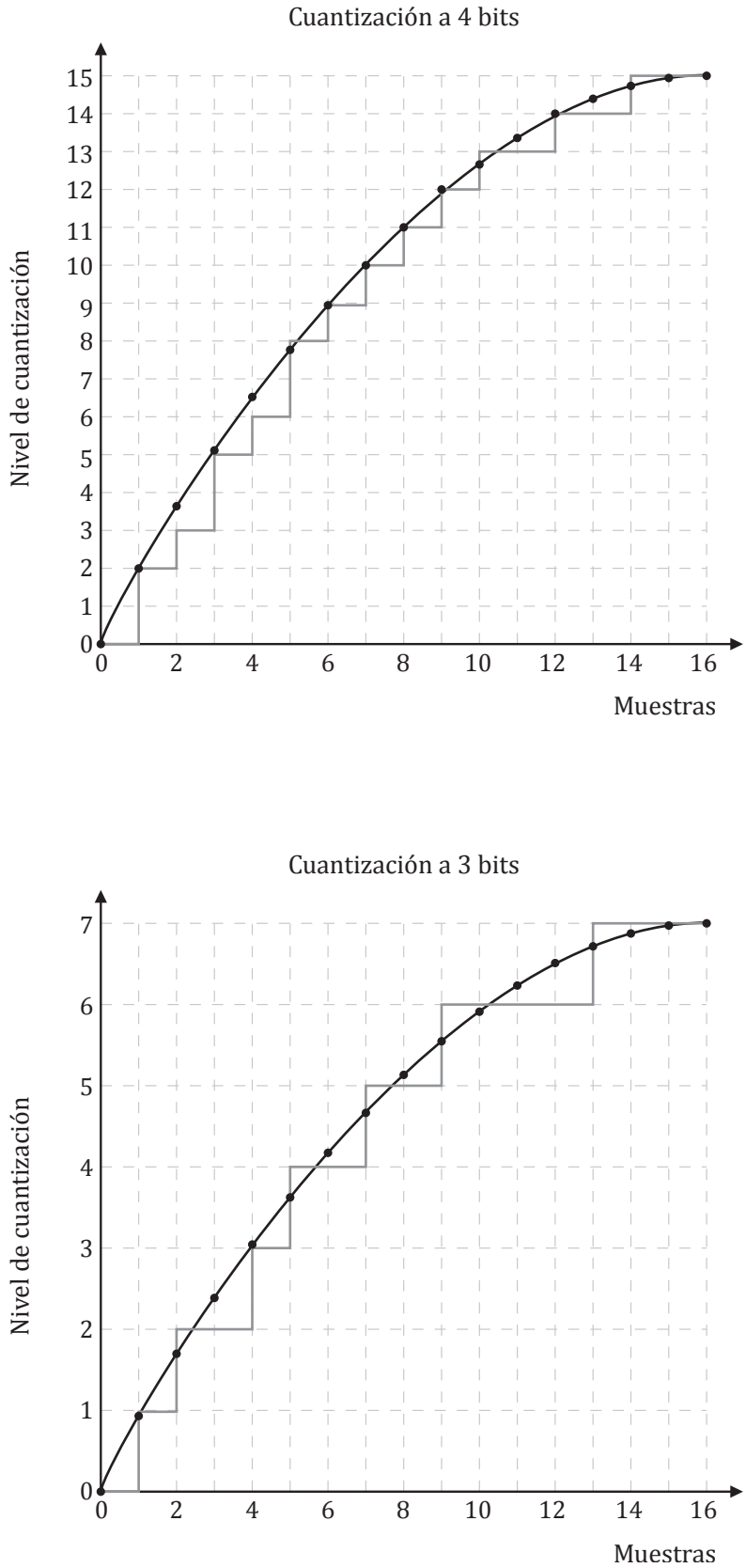
10.2.3 Cuantización

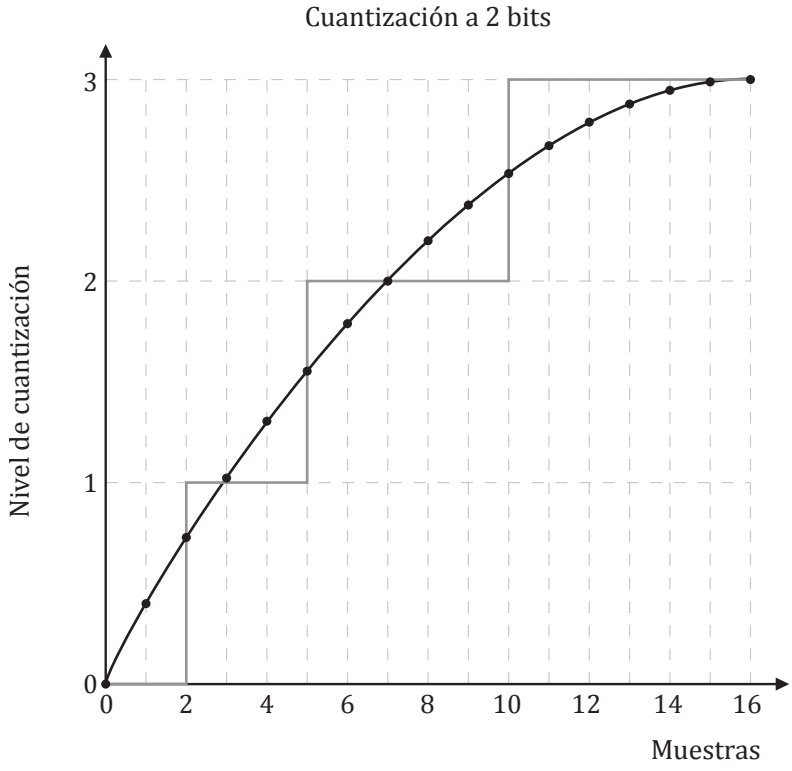
La cuantización es el proceso por el cual se comienza a discretizar la amplitud de las muestras obtenidas en el proceso de muestreo. Aquí, el rango de amplitud de entrada del convertidor analógico digital ($V_{REFH} - V_{REFL}$) se divide en intervalos de ΔV voltios. A este voltaje se le denomina «precisión de la conversión». El tamaño del intervalo depende del número de bits usados por el convertidor analógico digital, que se conoce como «la resolución r». El número de intervalos es equivalente a 2^r y el tamaño de cada intervalo ΔV es equivale a la siguiente fórmula:

$$\Delta V = \frac{(V_{REFH} - V_{REFL})}{2^r}$$

Conforme el número de bits r aumenta, la precisión de la conversión es mejor, ya que el intervalo ΔV es más pequeño y, por lo tanto, el error de cuantización es menor. En el gráfico 10.5 se observa una señal muestreada y cuantizada en tres niveles de resolución diferentes: 4 bits, 3 bits y 2 bits. Para el primer caso, el número de intervalos de cuantización es mayor ($2^4 = 16$), con lo cual el escalón que se observa en la gráfica izquierda del gráfico 10.5 representa la señal cuantizada y tiene una mejor aproximación a la señal original. Conforme el número de bits disminuye, los intervalos se reducen a 8 y 4 (3 y 2 bits, respectivamente), generando una señal escalón más espaciada y perdiendo la aproximación a la señal original (gráficas del medio y de la derecha del gráfico 10.5, respectivamente).

Gráfico 10.5 Diferencias entre la cuantización a 4 bits, 3 bits y 2 bits





Elaboración propia.

El proceso de cuantización se denomina «irreversible», ya que una vez realizado es imposible volver a obtener la información de la señal original. Una cuantización siempre presenta un error, conocido como el «Error de cuantización», que representa la diferencia entre el valor de amplitud real de la muestra y el valor asignado por el cuantizador. Para reducir este error, la precisión del ADC debe ser menor, lo cual implica un mayor número de bits de resolución. Para un cuantizador uniforme, el error de cuantización es ΔV/2.

10.2.4 Codificación binaria

Una vez que la señal ha sido cuantizada y cada muestra corresponde a un intervalo específico, se debe asignar un número binario de r bits a cada muestra. La forma de codificar dependerá del formato establecido por el ADC, el cual puede ser sin signo, complemento a 2 o en algún formato de Punto fijo donde la muestra se representa como número decimal.

Por ejemplo, si una muestra en un cuantizador cae en el intervalo de 3 a 4, se le asignará el código binario 0011_2 en formato sin signo, o el código 1100_2 que corresponde a -4 en complemento a 2. La codificación no cambia la magnitud de la señal digitalizada, tan solo varía su interpretación.

Una vez que la muestra queda codificada, es ubicada en el bus de salida del conversor analógico digital para que el controlador la pueda recoger a través del bus de comunicaciones. Una vez finalizado este proceso, es posible adquirir la siguiente muestra analógica.

10.3 El teorema del muestreo

Establecer el tiempo de conversión adecuado Tc para una señal analógica requiere no solo conocer las limitaciones del conversor analógico digital, sino también saber las características de la señal a digitalizar.

La mayoría de señales que se encuentran en la naturaleza son no determinísticas. Esto implica que no existe una ecuación matemática que pueda representar cada uno de los puntos que conforman la forma de onda. Todas estas señales deben ser aproximadas mediante la transformada de Fourier, la cual corresponde a una sumatoria de funciones senos y cosenos de diversas frecuencias y amplitudes. De esta manera, toda señal no periódica se aproxima a una suma de senos y cosenos de una frecuencia mínima f_{min} hasta una frecuencia máxima f_{max}. El ancho de banda de la señal se define como la diferencia $f_{max} - f_{min}$. Este parámetro corresponde a las frecuencias de la señal que contribuyen con una potencia significativa. El ancho de banda posee unidades de hercios.

Toda señal no determinística presenta un determinado ancho de banda. De acuerdo con la naturaleza de la señal (temperatura, voz, video, señal biomédica, etc.), esta tendrá un ancho de banda variable. Por ejemplo, la señal de voz presenta un ancho de banda de 4 kHz. Esto implica que una señal no determinística de la voz comprende frecuencias significativas hasta los 4 kHz. Por otro lado, una señal del corazón ECG usada en electrocardiografía presenta un ancho de banda de 250 Hz.

El teorema del muestreo de Nyquist postula que al momento de digitalizar una señal, la frecuencia de muestreo elegida debe superar al menos el doble del ancho de banda (B) de la señal. Esto garantiza una perfecta reconstrucción. Por tanto, la relación de Nyquist es la siguiente:

$$f_{muestreo} \geq 2B$$

Si no se respetara esta relación, el muestreo sería erróneo y se produciría un *Aliasing*, en el cual la señal generada digitalmente presentará características en forma diferentes a la señal original.

De esta manera, para muestrear la señal de voz se tendría que elegir al menos una frecuencia de muestreo de 8 kHz, y, para el caso de la señal ECG, esta frecuencia debería ser 500 Hz. Si se elige una frecuencia de muestreo mayor al doble del ancho de banda, esto definitivamente mejoraría el proceso de reconstrucción, presentando mejor nivel de detalle de la señal original. La contraparte es que la elevación de la frecuencia de muestreo reduce el tiempo de muestreo Tc de forma proporcional, lo cual resulta en una mayor cantidad de muestras por segundo. Si el controlador no posee la suficiente velocidad y capacidad de memoria, la cantidad de muestras digitales generadas por el ADC pueden resultar en un cuello de botella para el controlador.

La frecuencia de muestreo también se encuentra directamente relacionada con la tasa de bit. Este parámetro representa la cantidad de bits por segundo que genera el proceso de adquisición de una señal. Su unidad es el bit por segundo o bps y se calcula de la siguiente manera:

$$\text{tasa de bit} = r * f_{muestreo}$$

Por ejemplo, un ADC de 16 bits de resolución a una frecuencia de muestreo de 10 kHz produce una tasa de bit de 10 000×16 = 160 000 bps o 160 kbps, lo cual es una cantidad considerable de información. A mayor tasa de bit, se requerirá una mayor capacidad de recursos del controlador, tanto en velocidad como en espacio de memoria de datos para el almacenamiento de toda la información en un corto tiempo. Se debe de tomar en cuenta que el tiempo de muestreo dependerá no solo de la velocidad de conversión del ADC, sino también del tiempo de respuesta del controlador, ya que es este quien debe procesar cada muestra que es recibida del convertidor. El procesamiento más básico al menos consiste en recoger la muestra y almacenarla en una dirección de memoria de datos (porque es la que presenta un acceso más veloz, en comparación con una memoria EEPROM o FLASH).

10.4 El convertidor analógico digital del PIC18F4550

El convertidor analógico digital del PIC18F4550 posee 13 canales de entrada distribuidos en trece pines de E/S del microcontrolador para la versión de empaque DIP40. Este módulo se encarga de convertir una señal analógica a un valor digital de 10 bits de resolución. Dicha señal deberá estar delimitada por las tensiones de 5 V y 0 V o entre los rangos definidos por los pines V_{REFH} y V_{REFL}.

El módulo de conversión analógica digital se opera en base a cinco registros de función especial. Estos registros son los siguientes:

1. ADDRESH: parte alta del registro de resultado de conversión.
2. ADDRESL: parte baja del registro de resultado de conversión.
3. ADCON0: registro de control del modo de operación del ADC.
4. ADCON1: registro de control de configuración de los pines de entrada al ADC.
5. ADCON2: registro de control del reloj del ADC, la justificación del resultado y tiempo de adquisición.

El diagrama de bloques del convertidor analógico digital se muestra en el gráfico 10.6. Allí se puede observar que existen en total 13 canales de entrada al ADC, de los cuales solo uno de ellos puede ser seleccionado a la vez a través de los interruptores asociados a cada canal de entrada. El interruptor a cerrar contacto se selecciona mediante los bits CHS3-CHS0 del registro ADCON0. Solo un interruptor puede estar cerrado a la vez, así que, en caso de que se quieran digitalizar varias señales, la entrada del ADC tendrá que alternar los interruptores continuamente para cada señal de entrada. Esto, obviamente, generará un aumento significativo en el periodo de muestreo que deberá aplicarse a cada señal.

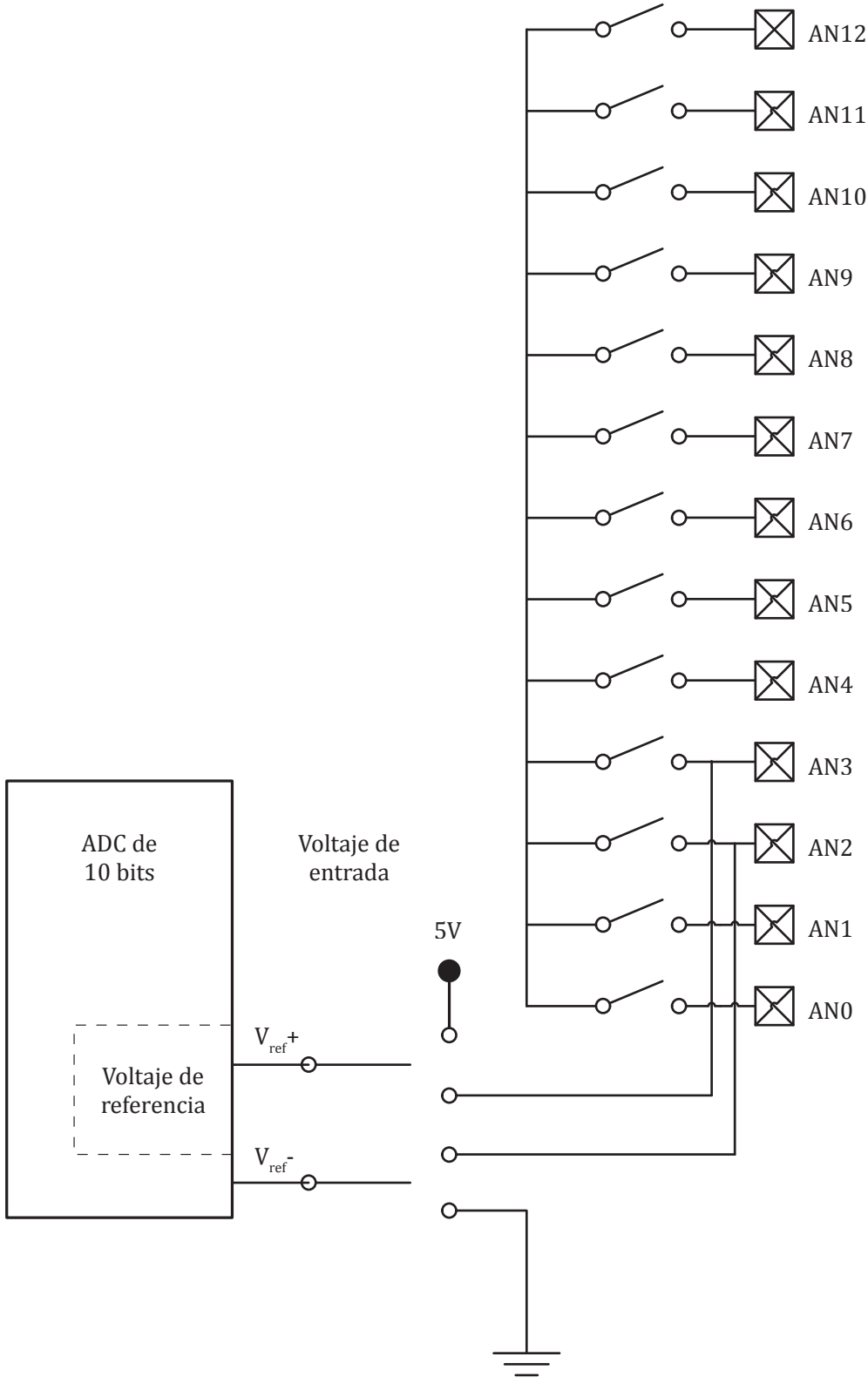
Adicionalmente, los pines AN2 y AN3 pueden operar como las entradas de referencia V_{REF}- y V_{REF}+, lo cual permite que la señal de entrada tenga otros niveles de voltaje de referencia máximos y mínimos diferentes de 5 V y 0 V, respectivamente. El cambio de voltaje de referencia se obtiene con los bits VCFG1 y VCFG0 del registro ADCON1. Cabe señalar que es sumamente importante que los pines usados como entrada para el ADC tengan sus respectivos bits en el registro TRISA, TRISE o TRISB con nivel '1', ya que, al estar configurados como entradas, el ADC podrá capturar la señal analógica de entrada de manera correcta y sin riesgo de producir cortocircuitos que puedan dañar al pin.

Adicionalmente a la configuración de los registros TRISA, TRISE y TRISB, es necesario indicar cuáles de los 13 pines de entrada al ADC tendrán un comportamiento analógico. En la mayoría de aplicaciones no se requiere conectar 13 señales analógicas al microcontrolador, sino una cantidad menor o a veces solo una. Ante esta situación, el programador puede configurar solo aquellos pines que va a utilizar como entradas analógicas, y el resto de pines los puede seguir empleando como entradas o salidas digitales.

El convertidor analógico digital trabaja en dos etapas: una de adquisición y otra de conversión. La primera etapa consiste en la carga del nivel de voltaje a la entrada del ADC. Si esta carga no se da en un tiempo adecuado, entonces el proceso de conversión dará un nivel incoherente acorde con el nivel de voltaje de entrada. Por tal motivo, el tiempo de adquisición para el caso del PIC18F4550 no debe ser menor que 6.4 µsegundos, valor que garantiza la correcta carga del condensador usado en la etapa de muestreo y retención. La segunda etapa está conformada por la conversión que se encarga de entregar los 10 bits correspondientes al valor de voltaje analógico en los registros ADRESH y ADRESL. La conversión de cada bit de la muestra demora un tiempo denominado T_{AD}, y la conversión de toda la muestra digital demora 11 T_{AD} (ya que un ciclo adicional es requerido para la descarga del condensador).

El tiempo de adquisición y el tiempo de conversión pueden ser controlados a través del registro ADCON2. El reloj de conversión normalmente utiliza la misma frecuencia de reloj del cristal externo usado para el procesador. También es posible utilizar la señal de reloj del circuito RC interno, lo cual permitiría que el ADC pueda operar en modo de bajo consumo. El divisor del reloj se configura para poder graduar los tiempos de adquisición y conversión. El tiempo de adquisición es un múltiplo de T_{AD}, mientras que el tiempo de conversión siempre será equivalente a 11 T_{AD}.

Gráfico 10.6. Diagrama de bloques del convertidor analógico digital del PIC18F4550



Elaboración propia.

El inicio de una adquisición y conversión se genera cargando el estado '1' en el bit GO/DONE. Este bit se mantendrá en nivel alto mientras duren estos procesos. Cuando la conversión haya finalizado (obteniéndose el resultado en los registros ADRESH y ADRESL), este bit se pondrá a nivel '0'. También es posible verificar la bandera ADIF del registro PIR1 para detectar el fin de la conversión. Además, se puede configurar una interrupción que indique la finalización de la conversión activando el bit ADIE del registro PIE1. En el gráfico 10.7 se observan los bits que componen al registro ADCON0.

Gráfico 10.7. El registro ADCON0

--	--	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON

Elaboración propia.

Las funciones de estos bits son las siguientes:

Bits CHS3-CHS0: permiten seleccionar el interruptor del canal que estará activo para conectar la señal analógica de un pin de entrada ANx a la entrada del convertidor analógico digital. Solo un interruptor puede estar activo a la vez, y este dependerá del estado de estos cuatro bits. En el gráfico 10.8 se puede ver la configuración de los bits de selección de canal analógico de entrada.

Gráfico 10.8. Bits de selección del canal analógico de entrada

CHS3	CHS2	CHS1	CHS0	Canal
0	0	0	0	AN0
0	0	0	1	AN1
0	0	1	0	AN2
0	0	1	1	AN3
0	1	0	0	AN4
0	1	0	1	AN5
0	1	1	0	AN6
0	1	1	1	AN7
1	0	0	0	AN8
1	0	0	1	AN9
1	0	1	0	AN10
1	0	1	1	AN11
1	1	0	0	AN12
1	1	0	1	No implementado
1	1	1	0	No implementado
1	1	1	1	No implementado

Elaboración propia.

Bit GO/DONE: este bit posee doble función. Si se escribe un nivel '1', un nuevo proceso de adquisición y conversión se iniciará. Posteriormente, este bit tendrá una operación en modo lectura. Cuando la conversión haya terminado y el bit se ponga automáticamente a nivel '0', se estará dando aviso de que el resultado de la conversión ya se encuentra listo en los registros ADRESH y ADRESL, y, por tanto, se puede iniciar una nueva conversión. Si el bit GO/DONE se pone a nivel '0' mediante un acceso por instrucción durante una conversión en curso, esta será interrumpida y anulada.

Bit ADON: este bit permite activar el módulo ADC y, de esta manera, puede operar. Si el bit se pone a nivel '0', el módulo ADC permanecerá desactivado.

Por otro lado, en el gráfico 10.9 se observa una imagen del registro ADCON1. Este presenta una función muy puntual, que tiene que ver con la configuración de los pines de entrada al ADC del PIC18F4550.

Gráfico 10.9. El registro ADCON1

--	--	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0

Elaboración propia.

La función de los bits es la siguiente:

Bits VCFG1-VCFG0: estos bits permiten configurar el voltaje de referencia que el ADC utilizará. Por defecto, $V_{REF}+ = 5$ V o VDD y $V_{REF}- = 0$ V, lo cual indica que el valor hexadecimal 0x000 corresponderá a 0 V y 0x3FF corresponderá a 5 V en la entrada. Sin embargo, es posible modificar las tensiones de $V_{REF}+$ y $V_{REF}-$ a un nivel menor que 5 V y mayor que 0 V, respectivamente. La mínima diferencia posible ($V_{REF}+ - V_{REF}-$) es de 1.8 V si VDD es menor que 3 V. Para una tensión de alimentación VDD mayor que 3 V, la mínima diferencia posible es de 3 V. En el gráfico 10.10 se pueden observar los modos de configuración para los voltajes de referencia según los bits VCFG1-VCFG0.

Gráfico 10.10. Bits de selección de los voltajes de referencia

VCFG1	VCFG0	$V_{REF}+$	$V_{REF}-$
0	0	VDD	GND
0	1	VDD	AN2
1	0	AN3	GND
1	1	AN3	AN2

Elaboración propia.

Bits PCFG3-PCFG0: estos cuatro bits permiten configurar los pines ANx que operarán como entradas analógicas y los que serán usados como entradas o salidas digitales. Esto se utiliza siempre

que algunos pines ANx sean usados para conectar señales analógicas y otros para manejar señales de entrada o salida digital. Los pines configurados como entradas analógicas solo pueden ser usados para este fin. Si un pin configurado en este modo es usado como entrada digital, siempre generará un nivel '0' como respuesta. En el gráfico 10.11 se observa la configuración de los canales de entrada. El símbolo D significa que el pin queda configurado como entrada o salida digital y A que el pin se configura como entrada analógica.

Gráfico 10.11. Configuración de los canales de entrada analógica

PCFG3-PCFG0	AN12	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
0000	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A	A	A	A
0111	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

Elaboración propia.

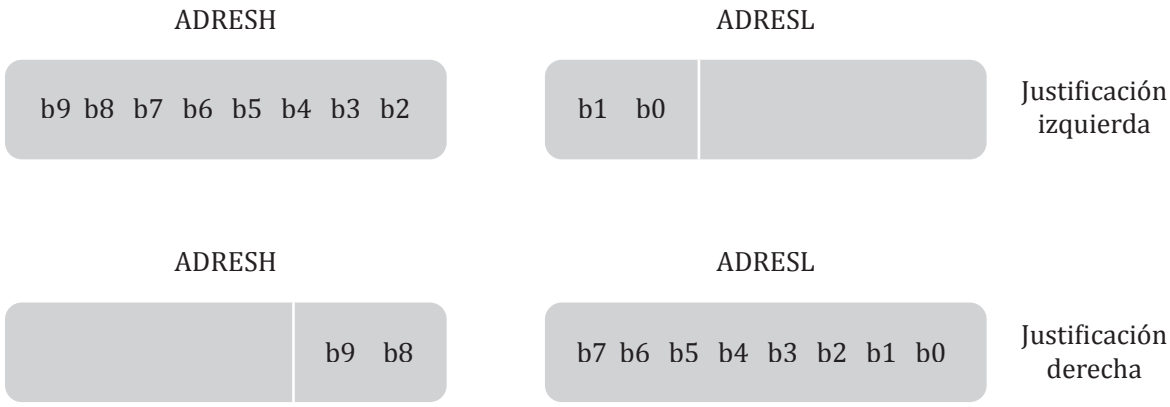
Finalmente, el registro ADCON2 permite configurar las opciones de reloj del ADC, el tiempo de adquisición y la justificación del resultado en los registros ADRESH y ADRESL. En el gráfico 10.12 se muestra el formato del registro ADCON2. La función de los bits del registro ADCON2 es la siguiente:

El bit ADFM: como se sabe, el convertidor analógico digital del PIC18F4550 presenta 10 bits de resolución; sin embargo, el microcontrolador presenta una arquitectura de 8 bits y por tanto, sus registros internos son de esta longitud. Para poder recibir un dato de 10 bits, el microcontrolador posee dos registros: ADRESH y ADRESL. El primero contiene la parte alta de la muestra y el segundo la parte baja. Siempre habrá seis bits de uno u otro registro que no tendrán uso. El formato para leer el código binario de la muestra de estos 10 bits puede ser de dos modos: con justificación a la izquierda (ADFM = '0') y con justificación a la derecha (ADFM = '1'), tal como se muestra en el gráfico 10.13. La justificación a la izquierda es ideal para trabajar con una resolución de 8 bits, ya que solamente se debe realizar la lectura del registro ADRESH para obtener la muestra a esta resolución.

Gráfico 10.12. El registro ADCON2

Elaboración propia.

Gráfico 10.13. Formato de salida del convertidor analógico digital en los registros ADRESH y ADRESL



Elaboración propia.

Los bits ACQT2:ACQT0: estos bits permiten configurar el tiempo de adquisición del proceso de conversión analógica digital de cada muestra. En muchas ocasiones conviene que este tiempo sea largo para que el consumo de energía del conversor sea menor. Cuando se configura este tiempo de adquisición, la espera se hace de manera automática durante la verificación del cambio de estado del bit GO/DONE. En el gráfico 10.14 se observan los posibles tiempos de configuración que se pueden establecer. La opción más baja (0 TAD) requiere que el programador genere una rutina de retardo mínimo en código de 6.4 µsegundos, que es requerida para que se cumpla la correcta conversión.

Gráfico 10.14. Bits de configuración del tiempo de adquisición

ACQT0	ACQT1	ACQT0	Tiempo de adquisición
0	0	0	0 T_{AD}
0	0	1	2 T_{AD}
0	1	0	4 T_{AD}
0	1	1	6 T_{AD}
1	0	0	8 T_{AD}
1	0	1	12 T_{AD}
1	1	0	16 T_{AD}
1	1	1	20 T_{AD}

Elaboración propia.

Los bits ADCS2:ADCS0: estos bits permiten configurar el tiempo de adquisición T_{AD} y, por lo tanto, posibilitan la configuración del retardo de conversión del ADC. Estos lapsos dependen de la frecuencia de reloj seleccionada y un divisor aplicado. Esto se puede observar en el gráfico 10.15.

Gráfico 10.15. Bits de selección del reloj de conversión

ADCS2	ADCS1	ADCS0	Frecuencia de conversión
0	0	0	Reloj derivado de un oscilador interno RC
0	0	1	$F_{RELOJ}/64$
0	1	0	$F_{RELOJ}/16$
0	1	1	$F_{RELOJ}/4$
1	0	0	Reloj derivado de un oscilador interno RC
1	0	1	$F_{RELOJ}/32$
1	1	0	$F_{RELOJ}/8$
1	1	1	$F_{RELOJ}/2$

Elaboración propia.

Para obtener T_{AD}, se debe seleccionar una de las opciones de reloj de conversión. Si F_{RELOJ} = 4 MHz y se ha seleccionado <ADCS2:ADCS0> = 110_2. Entonces:

$$T_{AD} = \frac{1}{4MHz / 8} = 2\mu seg$$

De esta manera, el tiempo de conversión será $11T_{AD}$, que equivale a 22 µsegundos, sumado al tiempo de adquisición, que dependerá del valor seleccionado en el gráfico 10.4.

En el programa 10.1 se muestra un ejemplo de adquisición de muestras digitales de una señal que se conecta al pin AN0/RA0 del PIC18F4550. Se utilizan solo los 8 bits más significativos de la muestra y se envían al Puerto D. Se selecciona un tiempo de adquisición de 4TAD y se configura el TAD = $1/(F_{RELOJ}/32)$.

Programa 10.1. Ejemplo de configuración del canal 0 del ADC para digitalización de un voltaje usando solo 8 bits de resolución

```
#include <P18F4550.INC>

        org 0x00
        goto MAIN
        org  0x08
        retfie
        org  0x18
        retfie
MAIN:
        clrf LATD
        clrf TRISD
        movlw 0xFF
        movwf TRISA
        movlw 0x0E
        movwf ADCON1        ;Solo AN0 opera como entrada analógica
        movlw .0
        movwf ADCON0        ;Se selecciona AN0 como entrada analógica
        movlw 0x15          ;Tiempo de adquisición = 4TAD. FOSC/4
        movwf ADCON2
        bsf   ADCON0,ADON
MUESTREO:
        bsf   ADCON0,GO
ESPERA:
        btfsc ADCON0,GO
        goto  ESPERA
        movf  ADRESH,W
        movwf LATD
        goto  MUESTREO
        END
```

10.5 Ejemplos de aplicaciones con el convertidor analógico digital del PIC18F4550

El convertidor analógico digital del PIC18F4550 es ideal para aplicaciones de adquisición de señales de un ancho de banda bajo, normalmente menor que 1 kHz. Es posible realizar la conexión de una gran variedad de sensores a esta tasa de muestreo. A continuación se presentan dos ejemplos aplicados que pueden servir como modelo para la configuración de cualquier otro sensor. Cabe mencionar que el con-

versor análogo digital no cuenta con la opción de activar un filtro *Antialising* previo a la entrada. Cualquier filtro de este tipo tendrá que ser conectado de manera analógica en la parte externa al pin ANx.

10.5.1 Diseño de un voltímetro digital usando el módulo LCD alfanumérico

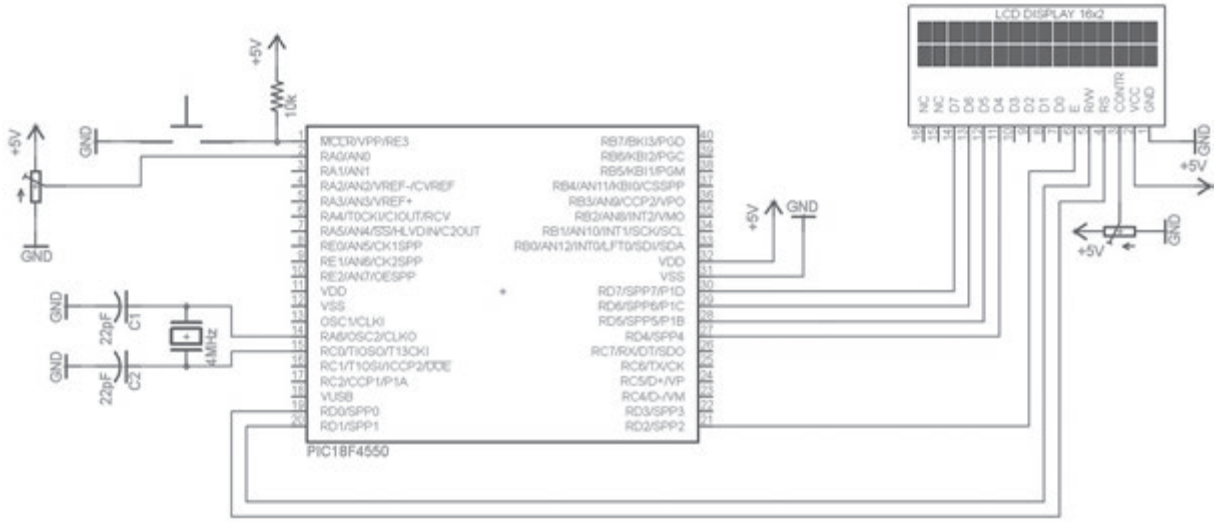
En el gráfico 10.16 se muestra el circuito esquemático con una propuesta de voltímetro digital. El sistema adquiere el voltaje generado por un potenciómetro que genera una tensión en un rango de entrada entre 0 a 5 V. Esto, obviamente, producirá un valor binario entre 0 y 1023, ya que se utilizarán los 10 bits de resolución total del ADC. La propuesta consiste en visualizar en la pantalla LCD el mensaje "Voltaje:" y luego el valor de voltaje con un dígito entero y dos dígitos decimales acompañados la letra V (unidad de voltios) al final de la cifra. De esta manera, el LCD mostrará un mensaje que, para 0 V en la entrada, será: "Voltaje=0.00 V", y para el máximo valor será: "Voltaje=5.00 V".

Para el manejo de la pantalla LCD se presentaron una serie de rutinas en el capítulo 7, las cuales configuraban el controlador HD44780 en lenguaje ensamblador. Estas rutinas se migraron a lenguaje C y se convirtieron en una librería. En el programa 10.2 se enlista el contenido del archivo LCD.h con los prototipos de las funciones que permiten manejar el módulo LCD alfanumérico en base a las conexiones mostradas en el gráfico 10.16. Se debe observar que el nombre de la función sugiere la actividad que realiza sobre la pantalla LCD.

Programa 10.2. Archivo LCD.h de la librería LCD

```
#define ON  0
#define OFF 1
#define RS    LATDbits.LATD0
#define RW  LATDbits.LATD1
#define E     LATDbits.LATD2

void POS_CURSOR(unsigned char fila, unsigned char columna);
void DISPLAY_ONOFF(unsigned char estado);
void CURSOR_HOME(void);
void CURSOR_ONOFF(unsigned char estado);
void ESCRIBE_MENSAJE(const char *cadena, unsigned char tam);
void ENVIA_CHAR(unsigned char dato);
void BORRAR_LCD(void);
void LCD_CONFIG(void);
void ENVIA_NIBBLE(unsigned char dato);
void ENVIA_LCD_CMD(unsigned char dato);
void LEER_LCD(void);
void BLINK_CURSOR(unsigned char val);
void GENERACARACTER(const unsigned char *vector, unsigned char pos);
```

Gráfico 10.16. Circuito esquemático para un sistema de adquisición de nivel de voltaje



Elaboración propia.

La función ENVIA_CHAR() recibe como parámetro de entrada un valor ASCII, el cual será impreso en la pantalla LCD. Para obtener el valor de voltaje a partir de la muestra de 10 bits leída de los registros ADRESH y ADRESL con una precisión de 2 decimales, se requiere hacer uso de una variable de tipo flotante de nombre «voltaje», capaz de obtener la magnitud del voltaje mediante la siguiente ecuación:

$$voltaje = \frac{5 \times (muestra)}{1023}$$

Dado que el valor de muestra nunca superará el rango de 1023, el máximo valor de voltaje será 5.00 y el mínimo 0.00. Como se puede ver, el dígito entero de la variable flotante solo podrá tener seis valores posibles: 0, 1, 2, 3, 4 o 5. Este dígito se extrae fácilmente, creando una variable de nombre aux tipo char de 8 bits y realizando la conversión de tipos de la siguiente manera:

```
aux = (unsigned char) voltaje
```

Con esta relación, la variable *aux* tomará únicamente la parte entera de la variable flotante *voltaje* e ignorará la parte decimal, la cual no puede ser asignada a *aux*. Posteriormente, para imprimir la cifra almacenada en *aux*, se deberá transformar a código ASCII. Como el valor es un número binario entre 0 y 5, entonces simplemente se sumará el código ASCII '0' a la cifra para realizar la conversión. Luego, se imprimirá en el LCD de la siguiente manera:

```
ENVIA_CHAR(aux+'0');
```

Para obtener los dos dígitos decimales en código ASCII se restará *aux* de *voltaje*, quedando la parte entera en 0. La parte decimal de la variable voltaje se multiplicará por 10, con lo cual el primer dígito decimal se convertirá a entero (por ejemplo, 0.89×10 = 8.9). Nuevamente, se repetirá el proceso de cambio de tipo de variable, asignando a *aux* la parte entera de voltaje; por lo tanto, se extraerá el

primer decimal. Luego, se volverá a restar *aux* de *voltaje* y a multiplicar por 10 este último, para que se pueda obtener el segundo decimal. Este proceso se repetirá cada 100 milisegundos, que será el tiempo de actualización del voltaje de entrada en la pantalla LCD. En el programa 10.3 se muestra la solución a este problema. Se debe tomar en cuenta que se utiliza una frecuencia de reloj de 48 MHz.

Programa 10.3. Algoritmo para mostrar el nivel de voltaje de entrada al canal 0 en un LCD alfanumérico

```
#include <xc.h>
#include <plib/delays.h>
#pragma config FOSC       = XTPLL_XT
#pragma config PLLDIV     = 1
#pragma config CPUDIV     = OSC1_PLL2
#pragma config PWRT       = ON
#pragma config WDT        = OFF
#pragma config LVP        = OFF
#pragma config BOR        = OFF
#pragma config DEBUG      = ON
#pragma config VREGEN     = OFF
#pragma config PBADEN     = ON
#pragma config MCLRE      = ON
#include "LCD.h"
unsigned int resultado;
float voltaje;
unsigned char aux;
void main()
{
        TRISA = 0xFF;
        TRISD = 0x00;
        ADCON1 = 0x0E;
        ADCON0 = 0;
        ADCON2 = 0x91;              //4TAD, FOSC/64, justificación a la derecha.
        LATD = 0;
        Delay1KTCYx(180);           //Retardo de 15 milisegundos.
        LCD_CONFIG();
        ESCRIBE_MENSAJE("Voltaje:",8);
        ADCON0bits.ADON = 1;
        while(1)
        {
                ADCON0bits.GO = 1;
                while(ADCON0bits.GO);
                resultado = 0x03 & ADRESH;
```

```
                resultado<<= 8;
                resultado +=ADRESL;
                voltaje = (float)resultado;
                voltaje = (voltaje*5)/1023;
                aux = (unsigned char)voltaje;
                POS_CURSOR(1,8);
                ENVIA_CHAR(aux+'0');
                ENVIA_CHAR('.');
                voltaje = voltaje - aux;
                voltaje = 10*voltaje;
                aux     = (unsigned char)voltaje;
                ENVIA_CHAR(aux+'0');
                voltaje = voltaje - aux;
                voltaje = 10*voltaje;
                aux    = (unsigned char)voltaje;
                ENVIA_CHAR(aux+'0');
                ENVIA_CHAR('V');
                Delay10KTCYx(120);     //Retardo de 100mseg
        }
}
```

Véase que para reconstruir el valor numérico de 10 bits se realiza la lectura del registro ADRESH y luego se desplazan los 2 bits más significativos de la muestra ocho posiciones a la izquierda. Finalmente, se suman los 8 bits menos significativos en la variable resultado, quedando la cifra de 10 bits completa. Para esto es necesario que la justificación de salida del ADC sea a la derecha (bit ADFM = '1').

Es preciso que el contenido de la variable resultado se asigne a la variable flotante *voltaje*, para que la división entre 1023 se pueda realizar con precisión decimal. Si la variable resultado se dividiera entre 1023, entonces el resultado de la operación sería determinado por este tipo de variable (unsigned int), lo cual generaría el resultado entero sin decimales. Sin embargo, en las nuevas versiones del compilador XC8 este paso ya no es necesario.

Al finalizar el programa, la pantalla LCD tendrá el formato mostrado en el gráfico 10.17, siempre y cuando la tensión conectada al pin AN0 fuera de 3.28 V. La precisión del voltaje puede aumentar si se utilizan 4 o 5 decimales adicionales. Uno de los inconvenientes de este programa es el hecho de que se debe realizar una operación de división y multiplicación de números flotantes, lo cual tiene una carga computacional considerable. Sería recomendable llevar esta etapa del programa a una rutina que permita obtener los resultados a partir de una tabla de 1024 valores y así evitar la operación de división. Esto sería computacionalmente más eficiente.

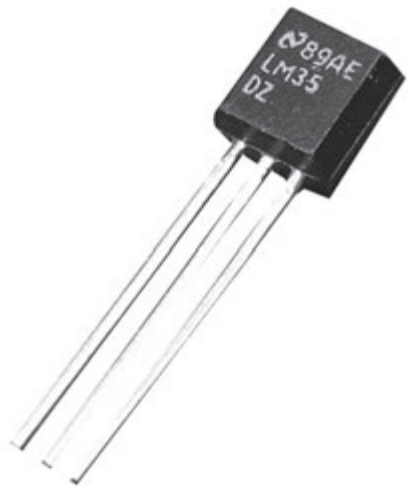
Gráfico 10.17. Mensaje mostrado en la pantalla LCD al momento de ingresar una tensión de 3.28 V al canal 0 del ADC

Elaboración propia.

10.5.2 Ejemplo de adquisición de temperatura usando el sensor LM35

El circuito integrado LM35 es un sensor de temperatura que se encuentra calibrado de fábrica para producir una tensión de salida proporcional a la temperatura medida. Este sensor es ideal para realizar aplicaciones de control o monitoreo de temperatura ambiente, ya que presenta un costo de adquisición bajo y es un dispositivo fácilmente acoplable a cualquier diseño electrónico. El rango de medición total es de -55 °C a 150 °C, y produce una salida de 10 mV/°C. En el gráfico 10.18 se muestra la imagen del sensor en empaque TO-92. También es posible conseguir este integrado en otros empaques como TO-220 o SO-8 de montaje superficial.

Gráfico 10.18. Sensor de temperatura LM35 en empaque TO-92



Fuente: www.spikenzielabs.com

En el gráfico 10.19 se muestra el circuito esquemático propuesto para el sistema de adquisición de temperatura. El sensor se conecta al canal 0 del ADC del PIC18F4550. La temperatura se mostrará en la pantalla LCD con precisión de 1 °C. La conexión del LM35 se muestra en el gráfico 10.19. Al alimentar el sensor con 5 V y una referencia de tierra, el rango de medición del sensor se reduce de 2 °C a 150 °C. Para el primer valor de temperatura la salida del sensor será 20 mV, y para el último valor, de 1.5 V, respectivamente.

El objetivo del algoritmo a programar en el microcontrolador es convertir el nivel binario obtenido del ADC en los registros ADRESH y ADRESL en el valor de temperatura correspondiente. Dado que en el programa 10.3 se diseñó un voltímetro digital y se obtuvo el nivel de voltaje en una variable de nombre *voltaje*, entonces se puede tomar como referencia este código para utilizarlo en el sistema de adquisición de temperatura.

Si la variable *voltaje* tuviera el valor de 0.4578, esto implica que el voltaje arrojado por el sensor LM35 es de 457.8 mV, lo cual corresponde a una temperatura de 45.7 °C, que aparecería en la pantalla LCD como 45 °C. En otras palabras, el valor de temperatura se obtendrá de multiplicar la variable *voltaje* por un factor de 100. Luego, se extraerán los tres dígitos enteros en BCD para ser convertidos a ASCII y enviados a la memoria DDRAM del módulo LCD cada 100 milisegundos. En el programa 10.4 se muestra la solución propuesta.

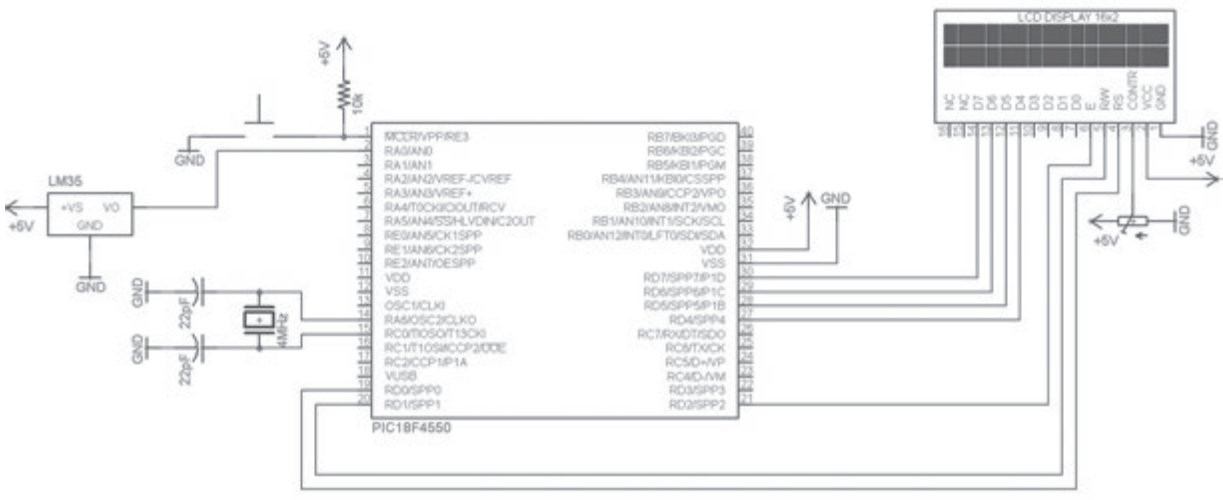
Programa 10.4. Adquisición de temperatura mediante el sensor LM35

```
#include <xc.h>
#include <plib/delays.h>
#pragma config FOSC   = XTPLL_XT
#pragma config PLLDIV = 1
#pragma config CPUDIV = OSC1_PLL2
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = ON
#pragma config MCLRE  = ON
#include "LCD.h"
unsigned int resultado;
float voltaje;
unsigned char temp;
void main()
{
        TRISA = 0xFF;
        TRISD = 0x00;
        ADCON1 = 0x0E;
        ADCON0 = 0;
        ADCON2 = 0x91;          //4TAD, FOSC/64, justificación a la derecha.
        LATD = 0;
        Delay1KTCYx(180);       //Retardo de 15 milisegundos
        LCD_CONFIG();
        ESCRIBE_MENSAJE("Temp:",5);
```

```
        ADCON0bits.ADON = 1;
        while(1)
        {
                ADCON0bits.GO = 1;
                while(ADCON0bits.GO);
                resultado = 0x03 & ADRESH;
                resultado<<= 8;
                resultado +=ADRESL;
                voltaje = (float)resultado;
                voltaje = (voltaje*5)/1023;
                voltaje = voltaje*100;
                temp    = (unsigned char)voltaje;
                POS_CURSOR(1,5);
                ENVIA_CHAR(temp/100+'0');
                ENVIA_CHAR((temp%100)/10+'0');
                ENVIA_CHAR((temp%100)%10+'0');
                ENVIA_CHAR('C');
                Delay10KTCYx(120); //Retardo de 100 milisegundos aproximadamente.
        }
}
```

Al finalizar el programa, la temperatura del sensor se actualizará en la pantalla LCD aproximadamente cada 100 milisegundos. El mensaje que se mostrará en pantalla tendrá el formato que se observa en el gráfico 10.20 si la temperatura del sensor fuera de 27 °C.

Gráfico 10.19. Circuito esquemático para un sistema de adquisición de temperatura ambiente



Elaboración propia.

Gráfico 10.20. Mensaje mostrado de la temperatura medida con el LM35

Elaboración propia.

Video

Diseño de un voltímetro digital haciendo uso del Conversor Análogo Digital del PIC18F4550.

http://bit.ly/1KWOGgK

Resolución de videos en HD

Preguntas de repaso

1. ¿Qué representa el ancho de banda de una señal?
2. ¿Qué significado tiene que la frecuencia de muestreo de una señal sea de 2 kHz?
3. ¿Cuál es la ventaja de realizar una digitalización con 12 bits de resolución sobre una que utiliza 8 bits?
4. ¿En qué se diferencia el tiempo de adquisición del tiempo de conversión?
5. Un sensor de dióxido de carbono tiene una precisión de 10 ppm (partes por millón) y arroja 2 mV por cada 10 ppm. Usted va a conectar el sensor a la entrada del convertidor analógico digital del PIC18F4550 que utiliza tensiones de referencia para su ADC de 5 V y 0 V. Es sumamente importante que el sistema digital a implementar mantenga la precisión de 10 ppm. ¿Podrá el ADC del PIC18F4550 ser usado para esta aplicación? Obtenga la mínima resolución necesaria del ADC y justifique su respuesta.
6. Un sensor de nivel de líquido de un tanque arroja 1 bit LSB por centímetro. La altura total del tanque es de 1023 centímetros. La salida del sensor se conecta al canal 0 del ADC del PIC18F4550. Este microcontrolador debe mostrar la altura en la pantalla LCD alfanumérica haciendo uso de la rutina de configuración del ADC interno. Diseñe un programa que muestre la altura del tanque en centímetros cada segundo en la pantalla LCD alfanumérica.

Respuestas

Respuestas del capítulo 10:

http://bit.ly/1Ng3gAl

Capítulo 11. El módulo mejorado de comunicación serial síncrona asíncrona (EUSART)

La gran mayoría de microcontroladores que abundan en el mercado poseen al menos un puerto de comunicación para poder establecer el intercambio de datos con el mundo exterior. Los PIC18F no son la excepción. ¿Qué sucedería en el ejemplo del capítulo anterior si se diseñara un sistema de adquisición de datos y se quisiera que la temperatura adquirida por el PIC18F4550 estuviera disponible en un servidor web o llegara a nuestro teléfono celular a través de un mensaje de texto? Pues bien, para solventar estas necesidades existen en el mercado diversas soluciones, como interfaces LAN Ethernet o módems GSM. Muchas de estas alternativas de solución presentan un puerto de comunicación serial asíncrono, que es el medio por el cual reciben la información de control y datos.

De esta manera, el PIC18F4550 cuenta con un módulo de comunicación serial asíncrono mejorado (EUSART, de las siglas en inglés *Enhanced Universal Synchronous Asynchronous Receiver Transmitter*) que permite la transferencia de datos a una serie de dispositivos externos, tales como interfaces LAN Ethernet, módems GSM, cámaras de fotos, pantallas LCD gráficas con malla táctil, computadores, etc. El módulo se denomina «mejorado» porque cuenta con algunas alternativas de operación que las versiones anteriores de la familia PIC18 no poseían, como un mecanismo de autodetección de velocidad de comunicación. El módulo EUSART también puede operar en un modo síncrono para la comunicación con algunos periféricos, como ADC, DAC o EEPROM, entre otros.

En este capítulo se mostrará el modo de comunicación asíncrona. Para configurar el módulo EUSART del PIC18F4550 se deben tener en claro ciertos aspectos básicos de la comunicación serial asíncrona y del estándar RS232 que es utilizado para la comunicación del sistema embebido con el computador. Luego, se explicará el modo de operación y configuración del módulo EUSART del microcontrolador, y, finalmente, se presentarán algunos ejemplos aplicados para demostrar la capacidad de operación de este módulo.

11.1 Nociones básicas de la comunicación serial

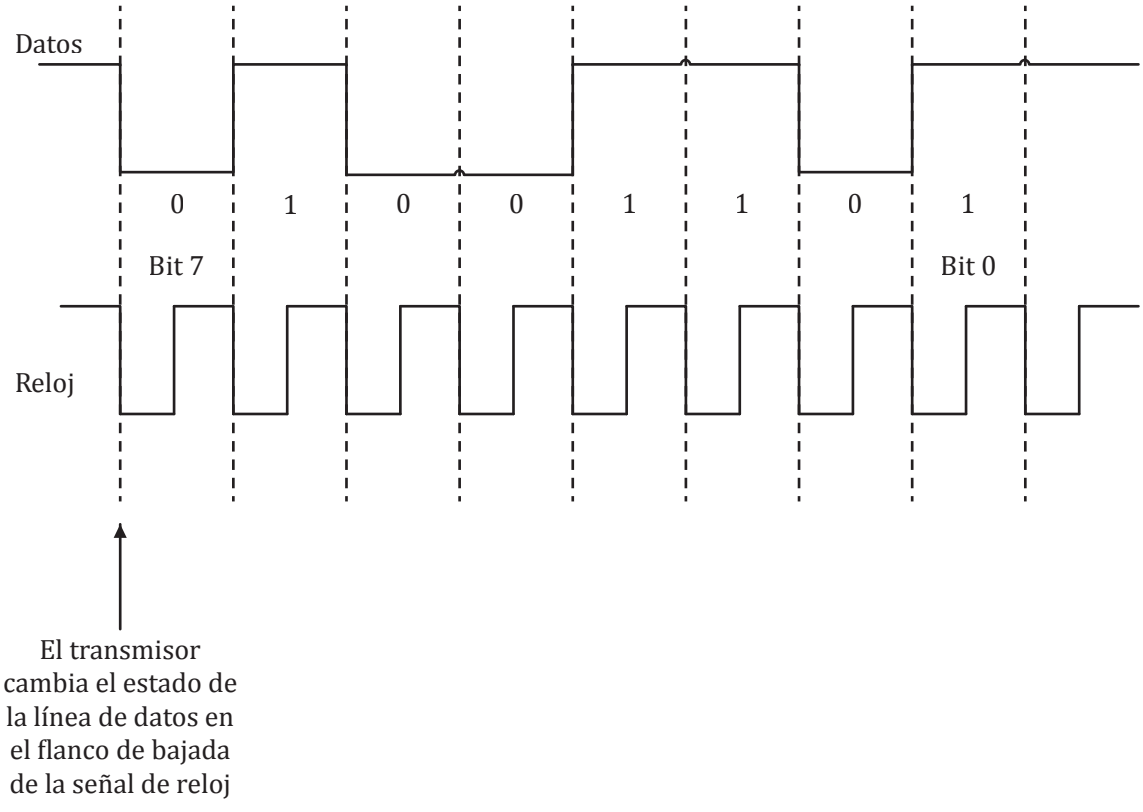
La comunicación serial se divide en dos formatos de comunicación: síncrona y asíncrona. La primera utiliza dos líneas de comunicación como mínimo, una para enviar los datos o bytes de información y la otra para la señal de reloj. La segunda solo contiene la línea de comunicación de datos. En el gráfico 11.1 se puede observar el formato de comunicación serial síncrona que es utilizada para los puertos de comunicación SPI e I^2C que se verán más adelante, en los siguientes capítulos del libro.

En dicho gráfico se muestran dos señales: reloj y datos. Cuando el transmisor genera el primer flanco de bajada o flanco activo de la señal de reloj, coloca el bit de información respetivo en la línea de datos. Por lo general, el bit más significativo se transmite primero en este tipo de comunicación. La señal de reloj, entonces, produce 8 pulsos o periodos de reloj, de los cuales el receptor deberá detectar los 8 flancos activos para realizar una lectura sobre la línea de datos. La frecuencia de reloj en este tipo de comunicación

puede variar, lo importante es que el receptor sea lo suficientemente rápido como para poder detectar cada flanco activo y realizar la lectura del bit en la línea de datos en un tiempo menor al periodo de reloj.

En el gráfico 11.2 se observa el formato de la comunicación serial asíncrono. La principal diferencia radica en que no existe una línea de reloj, solamente está presente la línea de datos. Además, se incluyen dos bits extra en la transmisión: el bit de inicio y el bit de parada. En total se transmiten 10 bits, aunque más adelante se verá que es posible agregar algunos bits adicionales. Cuando el transmisor se encuentra inactivo, la línea de datos permanece en un estado de espera, conocido como *Idle*, que se caracteriza por un nivel '1' lógico continuo. Cuando se inicia la transmisión de un byte de información, la línea de datos cambiará a un nivel '0', y se mantendrá en ese estado por un tiempo conocido como «el tiempo de bit» (T_b). Este bit es el bit de inicio o *Start*.

Gráfico 11.1. Sincronismo de la señal de reloj y de datos en la comunicación serial síncrona



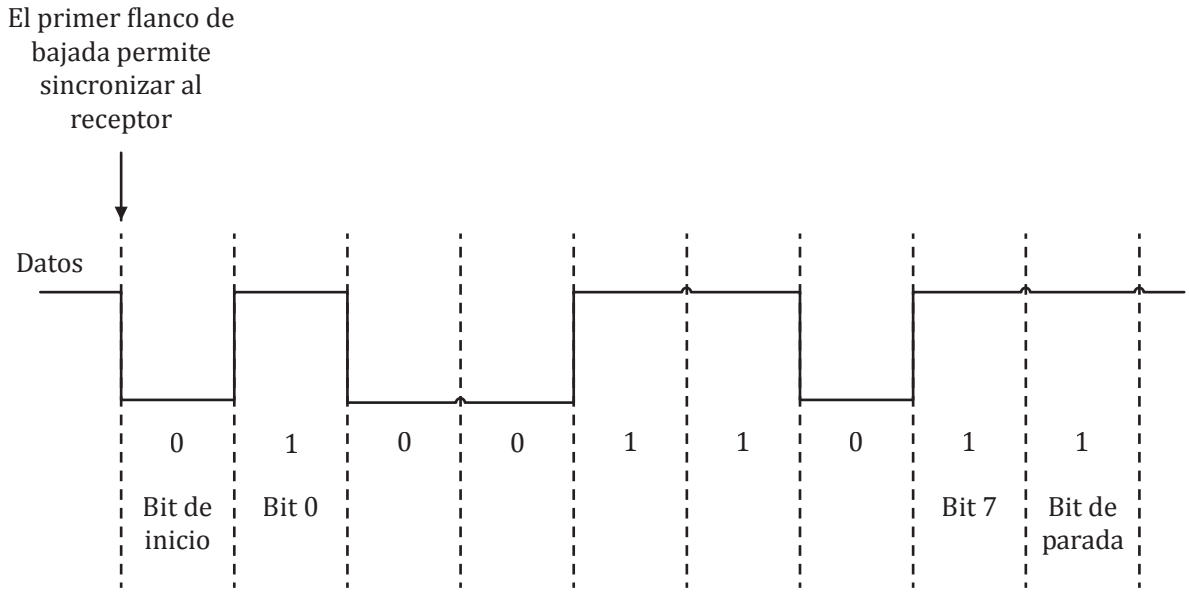
Elaboración propia.

El primer flanco de bajada de la señal de datos del estado de *Idle* al bit de inicio es crucial, ya que es este evento el que permite sincronizar al receptor. Posteriormente, se enviará el bit LSB de la trama de información hasta llegar al bit más significativo. Finalmente, se transmitirá el bit de parada o *Stop*, que siempre es un nivel '1'. Una vez terminada la transmisión de este bit, la línea de datos retornará al estado de *Idle*.

Al no existir una señal de reloj en la comunicación asíncrona, tanto el transmisor como el receptor deben sincronizar sus velocidades. Esto implica que el receptor deberá utilizar un reloj imaginario con un periodo T_b (tiempo de bit). Normalmente, el receptor muestrea la señal de datos a una frecuencia de $16T_b$ para detectar los estados de cada bit y determinar si no hay interferencia en la línea. Si las velocidades no son exactamente las mismas, aparecerá una probabilidad de error, con lo cual es posible que uno de los bits sea detectado erróneamente por el receptor.

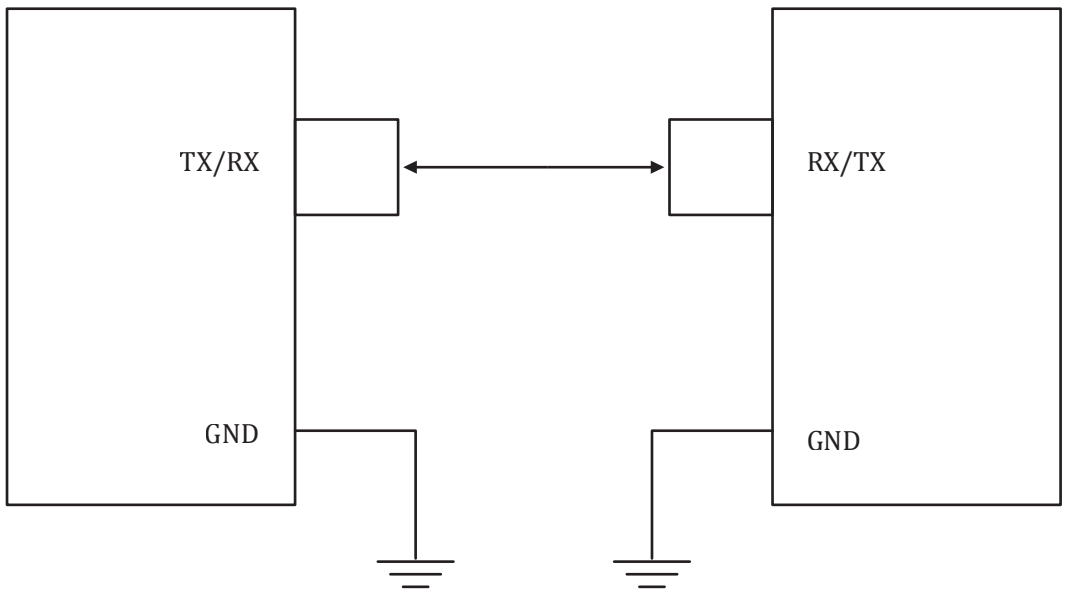
La comunicación serial también se divide en dos formatos según la bidireccionalidad de la información. Estos formatos son Half dúplex y Full dúplex. En el primer formato, el transmisor y el receptor intercambian roles enviando y recibiendo la información por la misma línea de comunicación, tal como se muestra en el diagrama del gráfico 11.3. El principal inconveniente aquí es el sincronismo entre ambos elementos de la comunicación, ya que, si ambos transmiten un dato a la vez, se puede originar una colisión, lo cual generaría una pérdida de los paquetes transmitidos. Este tipo de formato de comunicación se utiliza en el protocolo I^2C y en el protocolo SPI Microwire.

Gráfico 11.2. Señal de datos en la comunicación serial asíncrona



Elaboración propia.

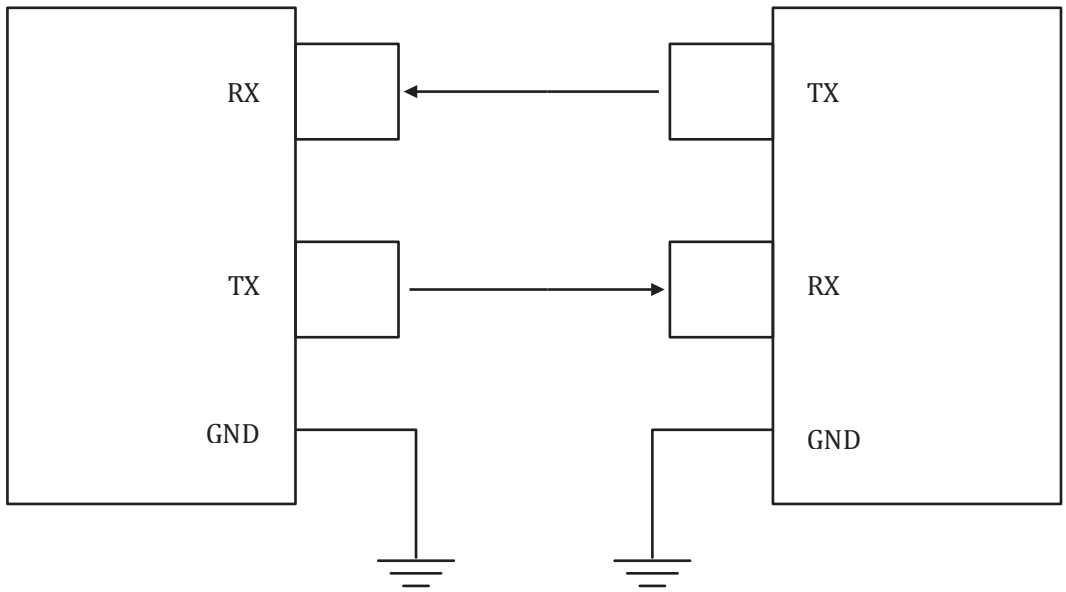
Gráfico 11.3. Esquema de la comunicación Half dúplex



Elaboración propia.

El formato Full dúplex permite la transmisión y recepción de datos en simultáneo. Esto se logra gracias a que existen dos líneas de comunicación independientes. Esto contribuye con una mayor velocidad de intercambio de información, ya que no es necesario enviar bits de control para generar el sincronismo entre transmisor y receptor. En la comunicación serial asíncrona del módulo EUSART se utiliza este formato de comunicación, al igual que en el protocolo SPI. El gráfico 11.4 muestra un esquema de la comunicación Full dúplex.

Gráfico 11.4. Esquema de la comunicación Full dúplex



Elaboración propia.

11.2 La tasa de bit y el formato NRZ

Durante la comunicación serial asíncrona, la ausencia de reloj obliga a fijar el periodo del reloj imaginario T_b. Este periodo determina la velocidad de comunicación que se mide en unidades de bits por segundo (bps) o también en baudios, donde 1 baudio es una unidad de medida que representa la transmisión de un símbolo por segundo en un canal de comunicación. Como en este caso cada símbolo es de 1 bit (ya que es la mínima unidad de información), se puede usar indistintamente el bps o el baudio para medir la velocidad de comunicación. La velocidad de comunicación o tasa de bit se puede calcular de la siguiente manera:

$$Tasa\ de\ bit = \frac{1}{T_b}$$

Si se tiene que en una comunicación serial asíncrona el tiempo de bit es de 1 milisegundo, entonces la tasa de bit será de 1000 bps o 1 kbps, o también de 1 kbaudio. Esto implica que será posible transmitir 1000 bits en un segundo. Cada trama estará dividida en 10 bits; por lo tanto, se transmitirán 100 tramas en cada segundo. Sin embargo, por cada trama hay 1 bit de inicio y bit de parada que no representan información. De esta manera, son 200 bits sin uso, lo que corresponde a una tasa de información equivalente de 800 bps. Es muy importante tomar en cuenta este análisis, ya

que muchas veces se fija la tasa de transmisión y esta resulta insuficiente para la cantidad de datos que se necesita enviar.

Para la comunicación serial asíncrona se utiliza el circuito integrado UART (Transmisor y receptor asíncrono universal, o *Universal Asynchronous Receiver Transmitter* en inglés), que se encarga de convertir una trama de entrada paralela en una trama serial y viceversa. El PIC18F cuenta con un UART interno capaz de generar las tramas de comunicación asíncronas de forma automática cuya velocidad de comunicación es programable. En la mayoría de dispositivos comerciales (computadoras, módems, pantallas LED gráficas, entre otros) que hacen uso de un módulo UART para su comunicación, las velocidades son estándares y el sistema embebido por lo general debe adecuarse a una de estas tasas para poder realizar una comunicación con un dispositivo externo. Las velocidades estándar se visualizan en el gráfico 11.5.

Gráfico 11.5. Velocidades estándar para la comunicación serial asíncrona

Velocidad
110 bps
300 bps
1200 bps
2400 bps
4800 bps
9600 bps
19 200 bps
38 400 bps
57 600 bps
115 200 bps
230 400 bps

Elaboración propia.

El formato NRZ (No retorno a cero, o *Non Return to Zero*, en inglés) implica que cada bit enviado en la comunicación serial asíncrona es representado por un nivel alto o bajo. Un bit '0' se representa con un nivel bajo en la señal, y un bit '1' lógico, con un nivel alto. La duración de cada estado es de T_b.

En la trama de comunicación serial asíncrona, los bits de datos pueden ser de 5, 6, 7, 8 y hasta 9. Adicionalmente, dada la probabilidad de error, muchas veces se agrega un bit de paridad P entre el bit MSB y el bit de parada, lo cual suma un bit adicional a la trama. El bit de paridad sirve para poder determinar en el receptor si la trama recibida ha llegado correctamente. La paridad puede ser de dos tipos: Par o Impar.

Se tiene una trama de información como la siguiente: 11100011_2. Como se puede notar, la cantidad de '1's en la trama es 5, es decir, hay un número impar de '1's. Entonces, si se elige una paridad Par, se tratará de que la trama presente un número par de '1's, incluyendo el bit de paridad. Entonces, P sería '1' para que la trama tenga seis '1'. Es decir, un número par. Si, por otro lado, la paridad fuera impar, el bit P sería '0', ya que los cinco '1's que presenta la trama corresponden a un número impar.

Cabe destacar que el bit de paridad será detectado por el controlador del receptor serial cuando realice el conteo de la cantidad de '1' en la trama recibida y no corresponda con el bit de paridad. Sin

embargo, muchos UART ya realizan este proceso de forma automática y generan un evento indicando el error en la trama.

11.3 El estándar RS232

El estándar RS232 es una interface que nació para ser utilizada en la comunicación entre computadoras haciendo uso de los puertos serie COM. El estándar tiene muchos años y en la actualidad los computadores no presentan estos puertos de comunicación. Sin embargo, el protocolo se sigue utilizando en una serie de dispositivos comerciales, tales como módems GSM, pantallas LCD gráficas, conversores USB-RS232, PLC, etc.

Este estándar fue diseñado para manejar la comunicación punto a punto de dos dispositivos a una distancia límite de 30 metros. Las líneas de comunicación en RS232 son desbalanceadas, es decir, todas tienen un nivel de referencia a tierra.

RS232 se utilizó mayormente para la comunicación entre un computador y un módem. La comunicación define dos elementos: el DTE (Equipo terminal de datos o *Data Terminal Equipment*, en inglés) y el DCE (Equipo de comunicación de datos o *Data Communication Equipment*, en inglés). El DTE normalmente es el controlador o computador en la comunicación y el DCE viene a ser el módem. En la práctica, para comunicar dos dispositivos es indiferente si uno de ellos es DCE o DTE; al final, la comunicación mantiene la misma jerarquía para ambos elementos de la interfaz.

Los DTE y DCE clásicos contienen un conector DB9 de nueve pines que cumplen con funciones específicas. Usualmente, el conector del DTE es de tipo macho y el DCE de tipo hembra. En el gráfico 11.6 se muestra una imagen de este conector.

Gráfico 11.6. Conector DB9 hembra

Fuente: http://www.eletrodex.com.br

La función de las nueve líneas del conector DB9 se muestra en el gráfico 11.7. Las líneas 2, 3 y 5 son fundamentales para la comunicación, mientras que las otras líneas se utilizan para el control de flujo y tienen una función específica cuando el DCE es un módem.

Gráfico 11.7. Pines del conector DB9 en el estándar RS232

Número de pin	Nombre	Señal generada por	Tipo de señal	Descripción
1	CD	DCE	Control	Detector de portadora.
2	RX	DCE	Datos	Recepción de datos.
3	TX	DTE	Datos	Transmisión de datos.
4	DTR	DTE	Control	Equipo terminal listo.
5	GND			Tierra común.
6	DSR	DCE	Control	Conjunto de datos disponible.
7	RTS	DTE	Control	Requerimiento de envío.
8	CTS	DCE	Control	Listo para enviar.
9	RI	DCE	Control	Indicador de timbrado.

Elaboración propia.

La línea TX transmite la información del DTE al DCE, mientras que la línea RX hace lo propio, pero en el sentido contrario. La línea GND debe unir el punto de tierra de DTE con la tierra de DCE. Las líneas restantes corresponden a señales de control cuyo uso específico determina la configuración de la comunicación entre un computador que hace la función de DTE y un módem que cumple la función de DCE. Estas líneas de control pueden ser utilizadas por la comunicación o también ignoradas si es que se implementa una interfaz que no requiere control de flujo.

El voltaje máximo y mínimo en el estándar RS232 varía de nivel, comparado con la señal TTL o CMOS mostrada en el gráfico 11.2. En el gráfico 11.8 se observan los niveles de voltaje para este estándar.

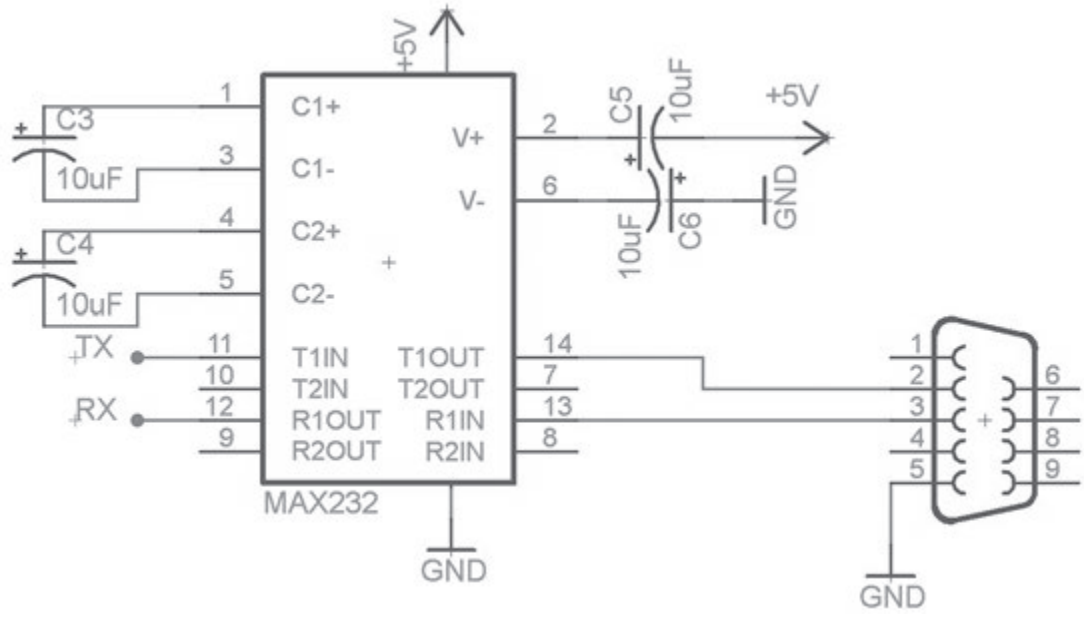
Gráfico 11.8. Niveles de voltajes para el estándar RS232

Nivel lógico	Voltaje
'0'	+5 V a +15 V
'1'	– 5 V a – 15 V

Elaboración propia.

Para conectar el módulo UART de un microcontrolador a una interfaz RS232, es necesario convertir los niveles de voltaje de las líneas del sistema embebido de TTL a los rangos de voltaje mostrados en el gráfico 11.8. Para esto, se necesita de un circuito integrado conversor de niveles que permita pasar de TTL a RS232 para la transmisión y de RS232 a TTL para la recepción. El circuito integrado más conocido para esta función es el MAX232 de la empresa MAXIM Dallas, cuya configuración se muestra en el gráfico 11.9. Este circuito integrado permite que el sistema embebido opere como un DCE. El MAX232 es un convertidor DC-DC que solo requiere de una tensión de alimentación de 5 V y cuatro condensadores electrolíticos de 1 a 10 µF conectados entres su pines.

Gráfico 11.9. Circuito integrado MAX232 polarizado y conectado al conector DB9



Elaboración propia.

El MAX232 recibe en sus pines 10 y 11 las señales TTL a transmitir y las transforma a niveles RS232 en los pines 14 y 7, respectivamente; mientras que los pines 8 y 13 reciben las señales en niveles RS232 y generan una señal equivalente TTL por los pines 12 y 9, respectivamente. La conversión de niveles del MAX232 también incluye la inversión de polaridad, donde 5 V se transforma en -15 V y 0 V en +15 V.

Si el sistema embebido trabaja con niveles CMOS de 3.3 V o 2.7 V, entonces el MAX232 resulta inapropiado. Para esto se tienen otras alternativas, como el circuito integrado MAX3243 o el MAX3232 de la misma empresa MAXIM Dallas.

11.4 El módulo EUSART del PIC18F4550

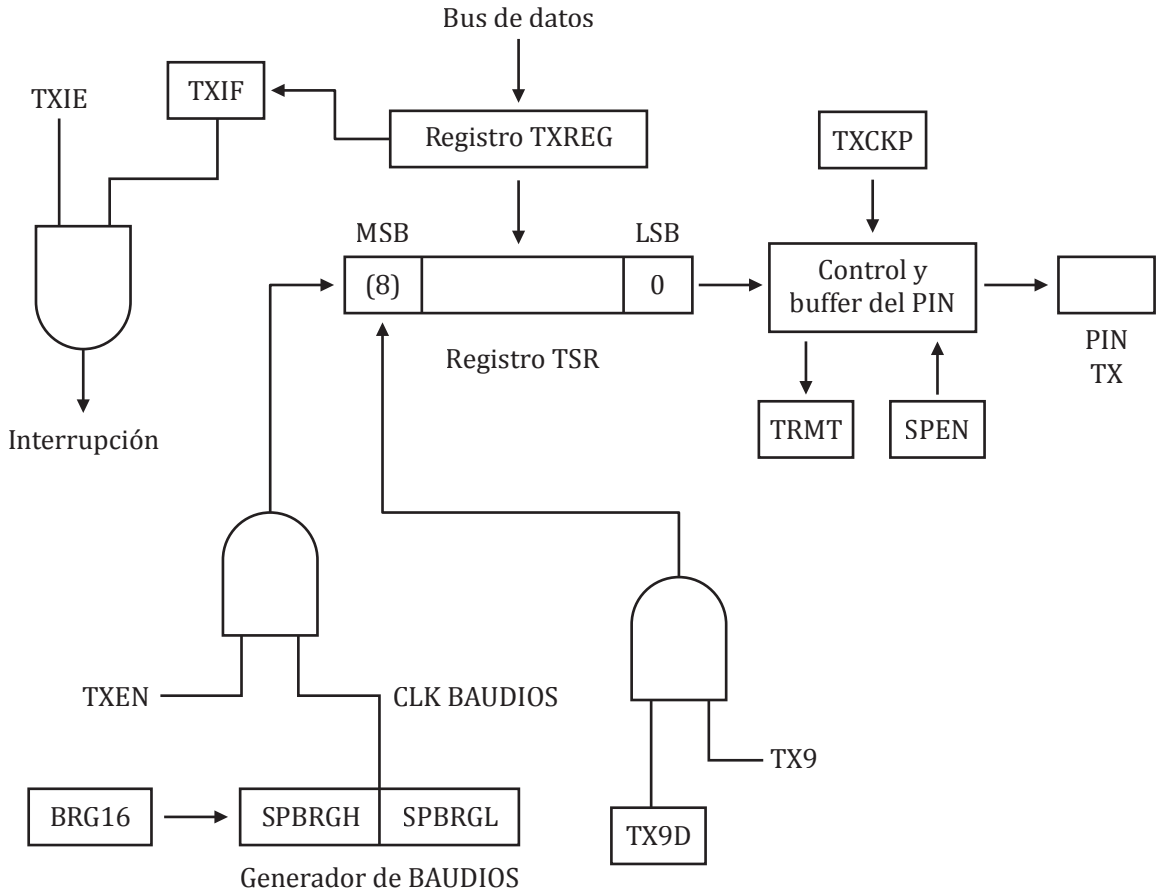
El PIC18F4550 cuenta con un módulo EUSART (del inglés *Enhanced Universal Synchronous Asynchronous Receiver Transmitter*) que corresponde a un módulo UART, que se configura también para generar una comunicación serial síncrona y que tiene algunas funciones especiales, como detección de un carácter *Break* de 12 bits y un modo de autodetección de velocidad, los cuales otorgan al módulo la denominación de «mejorado».

Los pines del Puerto C RC7 y RC6 cumplen con la función de ser la recepción (RX) y transmisión (TX) del PIC18F4550, respectivamente. Cuando el módulo EUSART se activa, la configuración del registro TRISC queda sin efecto alguno sobre estos pines.

En el gráfico 11.10 se muestra un diagrama de bloques de la etapa transmisora del EUSART. El proceso de transmisión comienza cargando un valor de 8 bits en el registro de función especial TXREG. Un ciclo de instrucción después de esta operación el contenido del registro TXREG se cargará en el registro de desplazamiento TSR, el cual comenzará a desplazar cada uno de los bits desde el LSB hasta

el MSB comenzando por un bit de inicio que siempre es '0'. El bit de bandera TXIF se pondrá a nivel '1' para indicar que el registro TXREG se encuentra vacío. Esto es importante, ya que el registro TSR demora un tiempo de $10T_b$ en desplazar todos los bits de la trama. Para esto, el bit de bandera TRMT se pone a '0', para indicar que el registro TSR se encuentra aún lleno. El programador deberá esperar a que este bit cambie a '1' para poder reescribir el registro TXREG.

Gráfico 11.10. Diagrama de bloques del transmisor EUSART



Elaboración propia.

Es posible también producir una interrupción habilitando el bit TXIE del registro PIE1, el cual generará el evento cada vez que el registro TXREG quede vacío. El bit TX9D puede cumplir con la función de operar como el noveno bit de datos si es que el bit TX9 se habilita con nivel '1'. Este bit también puede ser usado como paridad, pero la lógica de análisis de la cantidad de '1's en la trama se debe programar en el código, ya que el EUSART no cuenta con un circuito generador de paridad. La velocidad de transmisión del registro TSR estará comandada por la frecuencia de reloj del procesador y por un registro de 16 bits generador de velocidad SPBRG, que se divide en dos registros de 8 bits: SPBRGH y SPBRG. El bit TXEN habilita la señal de reloj para que el transmisor pueda operar. Si este bit se mantiene en nivel '0', la transmisión queda deshabilitada.

Los bits SPEN y TXCKP controlan el estado del pin de transmisión del microcontrolador. El bit SPEN habilita la salida de la señal por el pin TX y la recepción de señal por el pin RX. El bit TXCKP per-

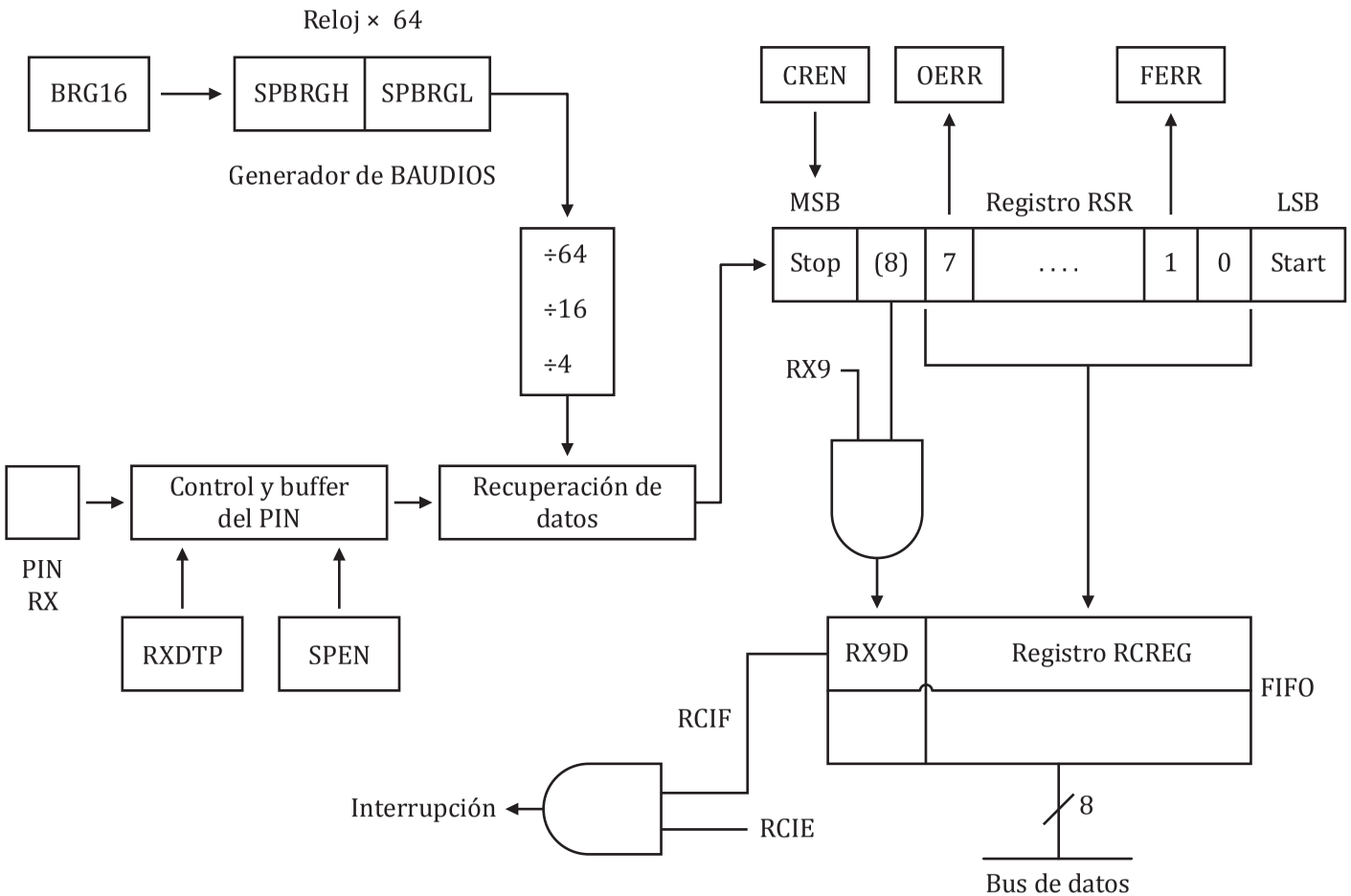
mite controlar la polaridad de la señal de salida. Como se sabe, en el formato RS232 la señal se invierte en polaridad. En el caso de utilizar un integrado como el MAX232, esta inversión resulta innecesaria.

Los pasos para configurar la transmisión de una trama asíncrona en el EUSART son los siguientes:

1. Inicializar los registros SPBRGH y SPBRG, así como el bit BRG16 que permite definir el modo de baja o alta velocidad.
2. Habilitar la comunicación asíncrona con el bit SYNC en nivel '0' y el bit SPEN en '1'.
3. Configurar la polaridad deseada en la transmisión con el bit TXCKP.
4. Si se desea habilitar la interrupción por transmisión, se debe poner el bit TXIE en '1'.
5. Si se desea transmitir una trama de nueve bits, se debe poner el bit TX9 a '1'.
6. Se debe habilitar la transmisión colocando el bit TXEN en nivel '1'.
7. El noveno bit de transmisión (si es que se ha configurado el modo de 9 bits en el paso 5) se cargará en el bit TX9D.
8. Cargar el dato a transmitir en el registro TXREG, lo cual inicia la transmisión.
9. Verificar el estado del bit TRMT para determinar el fin de la transmisión.

El diagrama de bloques del receptor del EUSART se muestra en el gráfico 11.11. Allí se puede observar que la señal de entrada ingresa por el pin de recepción RX. La lógica de control y buffer del pin es configurada a través de los bits SPEN y RXDTP. El primer bit habilita la recepción por el pin RX y el segundo bit controla la inversión de polaridad de la señal de entrada.

Gráfico 11.11. Diagrama de bloques del receptor del EUSART



Fuente: Hoja de datos del PIC18F4550.

Los registros SPBRGH y SPBRG controlan la velocidad de recepción de datos. Esta velocidad debe ser 16 veces mayor que la tasa de la señal de entrada, ya que se debe realizar un muestreo de 16 capturas de señal por cada tiempo de bit. Existe un divisor programable que permite configurar esta velocidad. La señal ingresa al registro RSR y se va desplazando de la posición MSB a la posición LSB. El bit CREN permite habilitar el funcionamiento del registro RCR para el desplazamiento de los datos.

Una vez que el registro RSR se ha llenado completamente con la trama recibida, el contenido se carga automáticamente en el registro RCREG. La bandera RCIF del registro PIR1 se pone automáticamente a nivel '1' para indicar que hay un dato recibido que se puede leer del buffer de recepción. Es posible habilitar una interrupción para este evento si el bit RCIE del registro PIE1 se pone a nivel '1'.

Se puede recibir un noveno bit de datos si es que se habilita el bit RX9 con el nivel '1'. El noveno bit no se detectará en el registro RCREG sino en el bit RX9D del registro RCSTA. Con el modo de recepción de 9 bits es posible fijar este bit como dirección, con lo cual se puede conectar el PIC18F4550 a una red de comunicación bajo el estándar RS485.

El registro RSR tiene dos bits de bandera indicadora de errores de recepción: FERR y OERR. El primer bit se pone a '1' para indicar un error de trama. Esto significa que el receptor no pudo detectar el bit de parada de la trama y, por lo tanto, se debería solicitar el reenvío de este dato. Por otro lado, el bit OERR indica que ha ocurrido una sobreescritura del registro RCREG, con lo cual el valor actual del registro RSR ha desplazado el contenido anterior de RCREG que nunca fue leído en el programa.

Los pasos para configurar la recepción asíncrona en el módulo EUSART son los siguientes:

1. Inicializar los registros SPBRGH y SPBRG para configurar la velocidad deseada. Los bits BRGH y BRG16 también deben configurarse de acuerdo con la velocidad.
2. Habilitar la comunicación serial asíncrona mediante el bit SPEN en nivel '1' y el bit SYNC en nivel '0'.
3. Definir si la señal recibida será invertida en polaridad configurando el bit RDXTP.
4. Si se desea habilitar la interrupción por recepción de trama, se deberá cargar el bit RCIE con '1'.
5. Si se desea recibir una trama de 9 bits, entonces el bit RX9 debe estar en nivel '1'.
6. Habilitar la recepción con el bit CREN en nivel '1'.
7. Esperar a que el bit RCIF sea '1'. Esto implica la recepción de una trama en el registro RCREG. Este evento también puede generar una interrupción si es que se habilitó el bit RCIE.
8. Una vez detectada la recepción, se deberá realizar la lectura del bit RX9D en el registro RCSTA si es que se ha seleccionado el modo de 9 bits de recepción.
9. Leer el registro RCREG.
10. Si ha ocurrido un error (verificando el estado de los bits FERR y OERR), realizar la operación respectiva y borrar estas banderas mediante la configuración del bit CREN en nivel '0'.

11.4.1 Los registros de configuración

El módulo EUSART posee tres registros de configuración donde se encuentran los bits mencionados en el subcapítulo anterior. Uno de ellos, el registro TXSTA, permite configurar el modo de transmisión. Por otro lado, el registro RCSTA configura el modo de recepción y el registro BAUDCON permite configurar la velocidad y algunas opciones en bajo consumo y de polaridad.

En el gráfico 11.12 se muestra una imagen del registro TXSTA y sus bits de configuración, y en el gráfico 11.13 se presentan las funciones de cada uno de estos bits.

Gráfico 11.12. Registro TXSTA y sus bits de configuración

Elaboración propia.

Gráfico 11.13. Bits de configuración del registro TXSTA

Bit	Nivel '0'	Nivel '1'
CSRC	En modo asíncrono, este bit no tiene aplicación.	En modo asíncrono, este bit no tiene aplicación.
TX9	La trama de transmisión se configura a 8 bits de datos.	La trama de transmisión se configura a 9 bits de datos.
TXEN	Transmisión desactivada.	Transmisión activa.
SYNC	Modo asíncrono.	Modo síncrono.
SENDB	Transmisión del carácter Break completada.	Enviar un carácter Break en la siguiente transmisión.
BRGH	Modo de baja velocidad.	Modo de alta velocidad.
TRMT	TSR lleno.	TSR vacío.
TX9D	Noveno bit de transmisión que puede ser usado como bit de paridad o de dirección.	

Elaboración propia.

Del mismo modo, en el gráfico 11.14 se muestra una imagen del registro RCSTA y sus bits de configuración, y en el gráfico 11.15 se ofrecen las funciones de cada uno de estos bits.

Gráfico 11.14. Registro RCSTA y sus bits de configuración

Elaboración propia.

Gráfico 11.15. Bits de configuración del registro RCSTA

Bit	Nivel '0'	Nivel '1'
SPEN	El puerto serial queda deshabilitado.	Se activa el puerto serie configurando los pines RX y TX como seriales.
RX9	La trama de recepción se configura a 8 bits de datos.	La trama de recepción se configura a 9 bits de datos.
SREN	En modo asíncrono no tiene uso.	
CREN	Deshabilita el receptor.	Habilita el receptor.

Bit	Nivel '0'	Nivel '1'
ADDEN	Deshabilita el modo de detección de dirección. El noveno bit puede ser usado como paridad.	Habilita el modo de detección de dirección. El noveno bit dispara una interrupción que habilita la carga del registro RCREG.
FERR	No hay error de trama.	Ha ocurrido un error de trama.
OERR	No ha ocurrido un error de sobreescritura.	Ha ocurrido un error de sobreescritura.
RX9D	Este bit puede ser usado como dirección o paridad.	

Elaboración propia.

Finalmente, el registro BAUDCON y sus bits de configuración se muestran en el gráfico 11.16, y las funciones de cada bit, en el gráfico 11.17.

Gráfico 11.16. Registro BAUDCON y sus bits de configuración

ABDOVF	RXDTP	RCIDL	TXCKP	BRG16	--	WUE	ABDEN

Elaboración propia.

Gráfico 11.17. Bits de configuración del registro BAUDCO

Bit	Nivel '0'	Nivel '1'
ABDOVF	No ha ocurrido un desbordamiento de los registros SBRG que actúan como contadores para la autodetección de velocidad.	Ha ocurrido un desbordamiento de los registros SBRG, con lo cual no se puede autodetectar la velocidad.
RCIDL	La señal en el pin de RX se encuentra en estado activo.	La señal en el pin de RX se encuentra en estado de IDLE.
RXDTP	La señal de entrada por el pin RX no se invierte de polaridad.	La señal de entrada por el pin RX se invierte de polaridad.
TXCKP	La señal de transmisión no se invierte de polaridad.	La señal de transmisión se invierte en polaridad.
BRG16	Solo el registro SPBRG es usado para la configuración de la velocidad.	Se utilizan los registros SPBGRH y SPBGRL para la configuración de velocidad.
NO IMPLEMENTADO		
WUE	El pin RX no es muestreado en modo de bajo consumo.	El pin RX será muestreado en bajo consumo.
ABDEN	No se habilita el modo de autodetección de velocidad.	Se habilita el modo de autodetección de velocidad. Se requiere la recepción de un byte 0x55 para la sincronización.

Elaboración propia.

11.4.2 El generador de baudios

El módulo EUSART cuenta con un generador de baudios basado en los registros SPBRGH y SPBRG y los bits BRGH y BRG16 que permiten configurar la velocidad deseada para la comunicación. Esta velocidad no solo depende de la configuración de estos registros y bits, sino también de la frecuencia de reloj externa. En el gráfico 11.18 se muestran las opciones de configuración de velocidad y las fórmulas usadas en cada una de estas. Cabe resaltar que el bit SYNC debe ser '0' para mantener la configuración del EUSART en modo asíncrono.

Gráfico 11.18. Fórmulas para el cálculo de la velocidad de comunicación

Bits de configuración		Modo del BRG	Fórmula para el cálculo de velocidad
BRG16	BRGH		
0	0	8 bits	$\frac{F_{OSC}}{64(SPBRG+1)}$
0	1	8 bits	$\frac{F_{OSC}}{16(SPBRG+1)}$
1	0	16 bits	
1	1	16 bits	$\frac{F_{OSC}}{4(SPBRG+1)}$

Elaboración propia.

Si se desea una velocidad de 115 200 baudios, se elige BRG16 en '1' y BRGH en nivel '0' y se opera a una frecuencia de reloj de 48 MHz, para hallar el valor que se debe ingresar a los registros SPBRGH y SPBRG se realiza el siguiente cálculo:

$$115200 = \frac{48000000}{16(SPBRG+1)}$$
$$SPBRG \approx 25.042$$

Dado que en el registro SPBRG no se puede ingresar un valor decimal, el resultado se redondea a 25 o 0x19. El hecho de que el resultado pierda los decimales implica que existirá un error en la configuración de la velocidad. Por ejemplo, si se reemplaza 25 en la ecuación se obtendrá lo siguiente:

$$velocidad = \frac{48000000}{16(25+1)}$$

$$velocidad = 115384.62\ baudios$$

Esto significa que la velocidad real generada por el microcontrolador no es exactamente 115 200 baudios, sino que presenta una desviación de 184.62 baudios. El error, entonces, será el siguiente:

$$error = \frac{(115385.62 - 115{,}200)}{115{,}200}$$

Un error de 0.16% se considera lo suficientemente bajo para que no se esperen errores de comunicación. Conforme este error aumente por encima del 2%, ya es posible que se puedan detectar errores y debería buscarse algún cambio en la configuración de la fórmula.

11.5 Ejemplos de aplicación con el EUSART del PIC18F4550

El módulo EUSART permite comunicar al microcontrolador con una gran variedad de dispositivos externos. A continuación se verán dos aplicaciones prácticas para la comunicación serial asíncrona. La primera utilizará la interfaz RS232 para comunicar al PIC18F4550 con un computador. Luego, la segunda aplicación utilizará el módulo EUSART haciendo uso de librerías en C para recibir datos de un sensor comercial de oximetría de pulso, a fin de diseñar un oxímetro de pulso haciendo uso también de la pantalla LCD alfanumérica.

11.5.1 Transmisión y recepción entre el PIC18F4550 y un computador

El objetivo de esta aplicación consiste en diseñar una interfaz de comunicación entre el microcontrolador y un computador. Estas interfaces son cruciales en un sistema de adquisición de datos, ya que permiten el envío de la información recopilada desde el sistema embebido a un computador, que puede almacenar estos datos y ponerlos a disposición de múltiples personas a través de una red o una base de datos.

El primer paso es dotar de un puerto serie RS232 a un computador actual. Hoy en día se trabaja con computadoras portátiles o de escritorio que solo cuentan con puertos de comunicación USB, Bluetooth, LAN Ethernet o WiFi, en su mayoría. Una alternativa de solución consiste en adquirir un conversor USB-Serial o utilizar un integrado que realice dicha conversión, como el FTDI232RL de la compañía FTDIchip. Obviamente, la solución más rápida es la compra de un cable convertidor. En el mercado existen diversas marcas, pero una de las más confiables y robustas es el convertidor TU-S9 de la marca Trendnet, que se puede conseguir fácilmente en el mercado nacional a un precio bastante accesible. En el gráfico 11.19 se puede observar una imagen de este producto.

Gráfico 11.19. Convertidor USB-RS232 modelo TU-S9 de Trendnet

Fuente: http://www.trendnet.com/langsp/products/proddetail.asp?prod=265_TU-S9

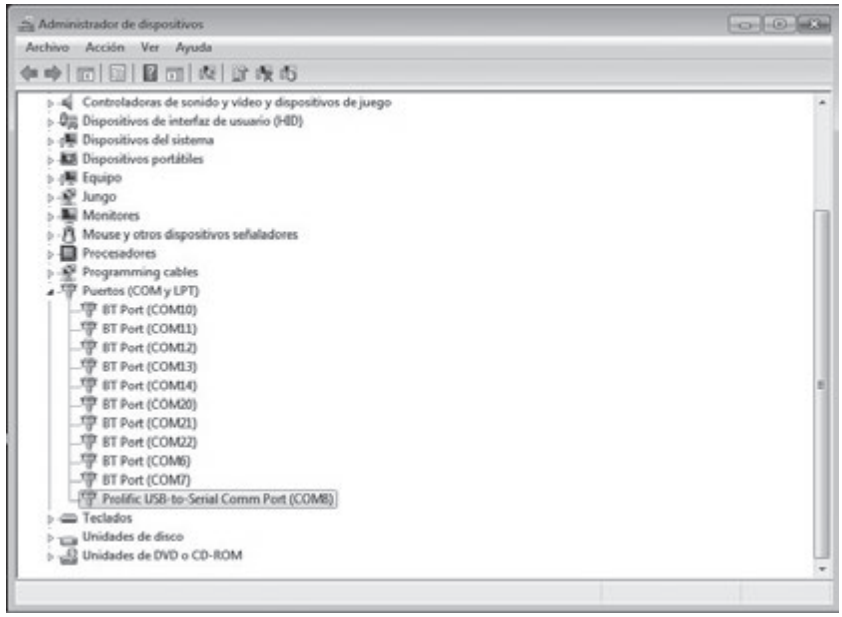
Al adquirir el producto, este estará en un estuche con un CD que contiene un *software* de instalación. Con él, el sistema operativo podrá detectar la configuración del *hardware* y generará un puerto serie virtual en el computador una vez que los *drivers* del controlador hayan sido correctamente instalados. De la misma manera, los *drivers* se encuentran disponibles en internet para su descarga gratuita. La ventaja de este producto es que puede operar en sistemas operativos antiguos, como Windows XP, y también en los actuales, que hasta la fecha son el Windows 7 y Windows 8. En el gráfico 11.20 se muestra una imagen del administrador de archivos de Windows luego de la instalación del TU-S9 en el sistema operativo Windows 7. Cabe mencionar que la asignación del puerto COM es automáticamente generada por el sistema operativo, pero el usuario la puede modificar a su conveniencia. En el gráfico 11.20 se puede ver que el puerto asignado es el COM8.

Para visualizar la información a transmitir y recibir en el computador existen diversos programas. Por ejemplo, se tienen herramientas de Microsoft como Visual C# .Net o Visual Basic .Net, que permiten realizar programas que operan sobre el puerto serie del computador. Sin embargo, estas herramientas requieren conocimiento de programación en lenguajes C# y Basic para poder ejecutar una comunicación. Una alternativa más sencilla es hacer uso del *software* Hyperterminal de Windows. Este programa venía incorporado con el sistema operativo hasta Windows XP. Las versiones de Windows 7 y Windows 8 no incorporan el Hyperterminal por defecto, pero existen en internet versiones que se pueden descargar de forma gratuita para estos sistemas operativos.

Una vez instalado correctamente el Hyperterminal, se debe abrir para crear una nueva sesión. Para esto, el usuario deberá realizar los siguientes procedimientos:

1. Seleccionar Inicio->Programas->Hyperterminal. Con esto aparecerá la ventana de diálogo del programa.
2. Escribir el nombre de la conexión y pulsar sobre el botón OK, tal como se muestra en el gráfico 11.21.

Gráfico 11.20. Administrador de dispositivos de Windows 7 mostrando la correcta configuración del convertidor USB-RS232

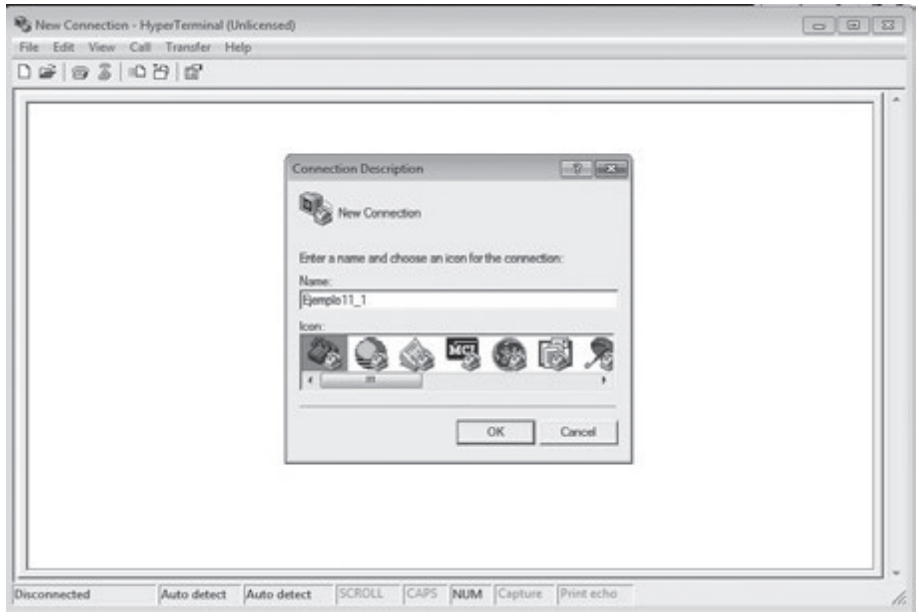


Elaboración propia.

3. Seleccionar el puerto de conexión. Para el caso de este ejemplo, el sistema operativo asignó el puerto COM8 al convertidor TU-S9. Una vez seleccionado, se pulsa sobre el botón OK. Este paso se muestra en el gráfico 11.22.
4. Luego, aparecerá una ventana de configuración de la velocidad, tipo de trama de transmisión y control de flujo. Esta ventana se observa en el gráfico 11.23. Aquí se deben configurar los siguientes parámetros:
   Velocidad: 9600 bps
   Bits de datos: 8
   Paridad: Ninguna
   Bits de parada: 1
   Control de flujo: Ninguno

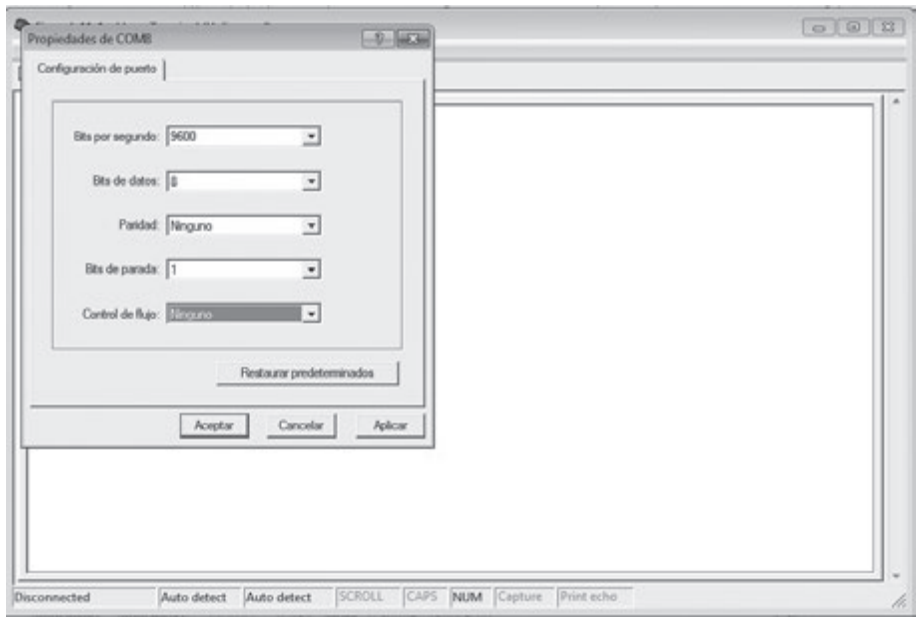
Finalmente, queda lista la ventana de comunicación del Hyperterminal, tal y como se observa en el gráfico 11.24. Todos aquellos caracteres que se escriban desde el teclado del computador serán transmitidos en código ASCII por el pin 3 del conector DB9 del TU-S9. En la pantalla solo se visualizarán los caracteres recibidos por el TU-S9 a través del pin 2 de su conector DB9.

Gráfico 11.21. Segundo paso para crear una nueva sesión en Hyperterminal



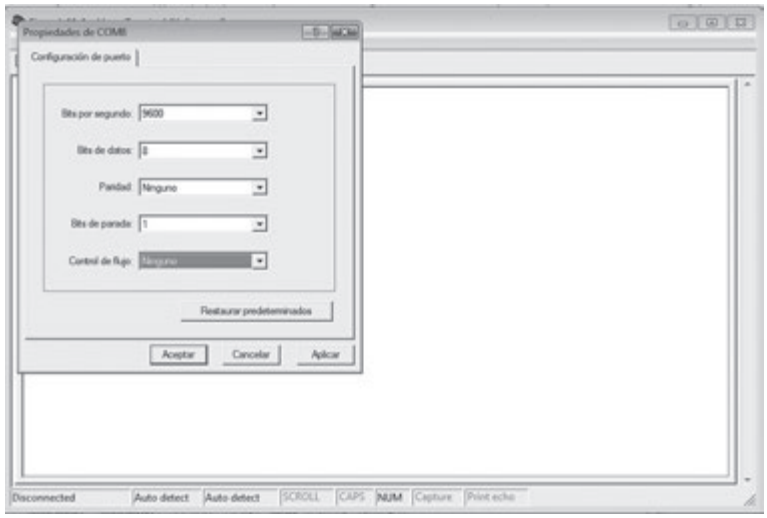
Elaboración propia.

Gráfico 11.22. Selección del puerto de comunicaciones COM asignado para el TU-S9



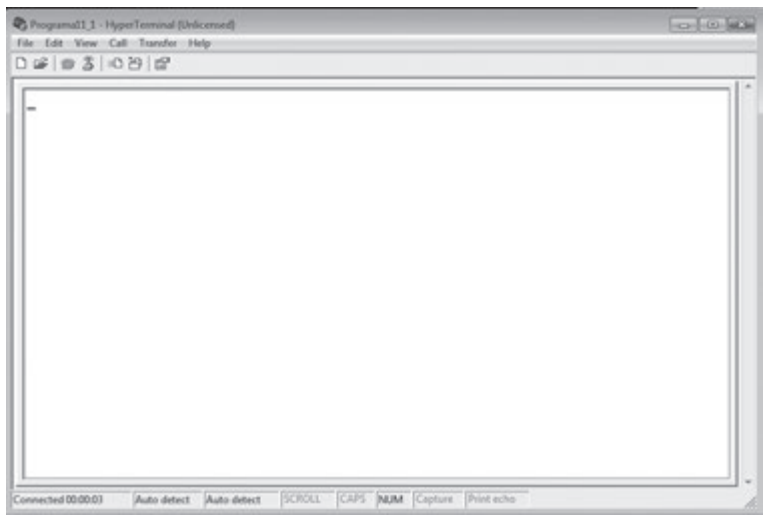
Elaboración propia.

Gráfico 11.23. Selección de las opciones de trama de datos de la comunicación RS232



Elaboración propia.

Gráfico 11.24. Ventana de comunicación del Hyperterminal



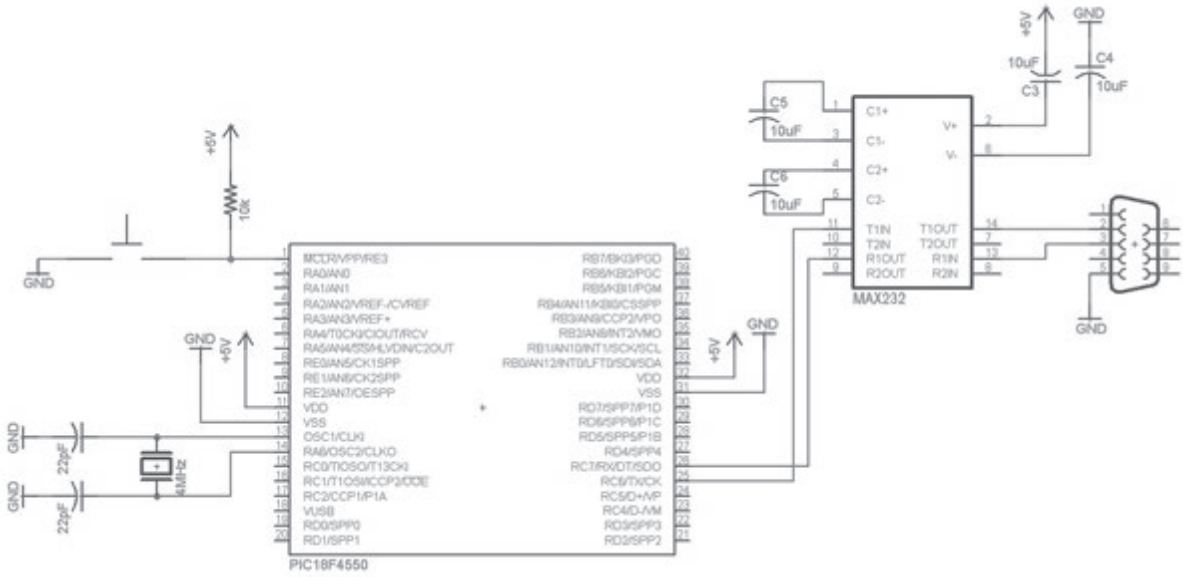
Elaboración propia.

Todos los caracteres a visualizar en la ventana de comunicación del Hyperterminal estarán en formato ASCII. Esto significa que solo los valores binarios menores que 0x80 tienen una representación alfanumérica en la pantalla. Existen también caracteres para el cambio de línea (0x0D o '\n') y retorno a la primera columna (0x0A o '\r'). De esta manera, el programador del PIC18F4550 deberá mantener los valores a transmitir en código ASCII, si es que se desea que la información pueda ser visualizada e interpretada en Hyperterminal.

La propuesta de solución para interconectar el PIC18F4550 al computador a través del cable TU-S9 se muestra en el circuito esquemático del gráfico 11.25. Allí se observa que el conector DB9 de tipo hembra del microcontrolador tiene el pin 3 como entrada de datos y el pin 2 como salida, cumpliendo con la función de DTE. De esta manera, el conector DB9 macho del TU-S9 se puede empalmar directamente al

conector DB9 hembra y la comunicación se realizará sin ninguna colisión. En caso de que el computador cuente con un puerto RS232, se deberá utilizar un cable directo para la conexión al circuito.

Gráfico 11.25. Circuito esquemático para la conexión del PIC18F4550 al computador



Elaboración propia.

El programa 11.1 realiza el eco de los datos transmitidos desde el *software* Hyperterminal. Cada vez que el usuario pulse una tecla de su computador con el *software* Hyperterminal, el código ASCII del valor de la tecla pulsada será transmitido al PIC18F4550. Este último recibirá el código binario y una interrupción se ejecutará cuando el dato esté listo para ser leído. En la rutina de interrupción se recogerá el dato y se retransmitirá al computador, donde el valor ASCII será visualizado.

El registro SPBRG se carga con el valor decimal 25, que corresponde a una velocidad de 9600 baudios para un cristal de 4 MHz. El programa simplemente se queda en un lazo indefinido a la espera de que ocurra una interrupción por recepción serial que se activa cuando la bandera RCIF es '1'.

Programa 11.1. Ejemplo de un DTE que realiza eco de los datos recibidos por el puerto RS232

```
#include <18F4550.INC>    ;Definicion de SFRs para el procesador
        org 0x00
        goto MAIN
        org 0x08
        goto RUT_INT
        org 0x018
        retfie
        org 0x20
```

```
MAIN:
        movlw .25
        movwf SPBRG             ;Configurando 9600 baudios para un cristal
        clrf  SPBRGH            ;de 4MHz.
        bcf   TXSTA,SYNC        ;Modo asíncrono.
        bsf   TXSTA,BRGH
        bcf   BAUDCON,BRG16
        bsf   RCSTA,SPEN        ;Habilitar módulo EUSART.
        bsf   TXSTA,TXEN        ;Habilitar la transmisión.
        bsf   PIE1,RCIE         ;Habilitar interrupción por recepción.
        bsf   RCSTA,CREN        ;Habilitar el receptor serial.
        bsf   INTCON,PEIE       ;Activar la máscara de interrupción de periféricos.
        bsf   INTCON,GIE        ;Activar la máscara global de interrupciones.
LAZO:
        goto  LAZO
RUT_INT:
        btfss PIR1,RCIF         ;Verificar bandera de recepción del EUSART.
        retfie
        bcf   PIR1,RCIF         ;Borrar la bandera de recepción.
        movf  RCREG,W           ;Leer el dato recibido.
        movwf TXREG             ;Transmitir el mismo dato.
ESPERA:
        btfss TXSTA,TRMT        ;Esperar hasta que TSR se encuentre vacío.
        goto  ESPERA
        retfie                  ;Retornar de la interrupción.
        END
```

Al instalar el programa, el usuario podrá visualizar en la pantalla del Hyperterminal los caracteres que escribe. Si se prueba retirando el cable o desenergizando el PIC18F4550, se podrá observar que los caracteres no aparecerán más. Como ejemplo, se puede escribir el mensaje «Hola Mundo!!» y verificar el resultado, tal como aparece en el gráfico 11.26.

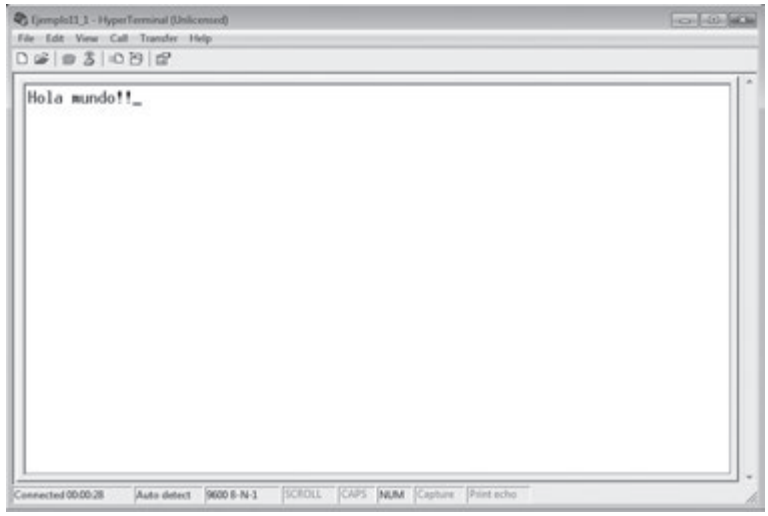
11.5.2 Comunicación con un oxímetro de pulso NONIN modelo IPOD 3212

La oximetría de pulso es un mecanismo de monitoreo biomédico muy importante, ya que permite conocer acerca del correcto transporte de oxígeno en las arterias del organismo. La concentración de oxígeno que se encuentra en la hemoglobina de la sangre es vital en el funcionamiento de las células que conforman los tejidos de cada órgano del cuerpo humano. Durante la ausencia de oxígeno por un tiempo prolongado, las células mueren. Así, el correcto transporte de oxígeno a las células es un indicador fundamental de la salud de un paciente.

La oximetría de pulso entrega tres tipos fundamentales de información. Dos de ellos son datos numéricos, como el pulso cardiaco y el nivel de saturación de oxígeno (SPO_2). El tercer parámetro es una gráfica que

evoluciona en el tiempo, conocida como la «onda fotopletismográfica»[47]. El pulso cardiaco permite conocer la cantidad de pulsaciones o ciclos cardiacos que el corazón realiza en un minuto. Las unidades de este parámetro se dan en pulsos por minuto (bpm, del inglés *Beats Per Minute*). El nivel de saturación de oxígeno es una unidad que permite conocer de forma estimada la relación entre la hemoglobina oxigenada y los demás tipos de hemoglobina presentes en la sangre arterial (que no transporta oxígeno). Este parámetro no tiene unidades y se mide en porcentaje (%), donde el máximo valor corresponde al 100% (sangre con un alto índice de hemoglobina oxigenada) y el valor más bajo puede estar bordeando el 30% (sangre con una deficiencia abrupta de hemoglobina oxigenada). La onda fotopletismográfica es un tercer parámetro que no necesariamente aparece en todos los oxímetros de pulso[48]. Esta onda posee una forma característica que permite notar detalles como el tipo de flujo sanguíneo del paciente y la forma de inyección de oxígeno en la sangre. Normalmente, se requiere de una pantalla gráfica en el oxímetro de pulso para poder visualizar esta onda. En el gráfico 11.27 se observa una forma de onda típica de una señal fotopletismográfica obtenida del autor de este libro.

Gráfico 11.26. Mensaje de eco retransmitido por el PIC18F4550



Elaboración propia.

La adquisición de la información de oximetría de pulso se basa en el uso de un sensor o clip. Este clip posee dos diodos emisores de luz (LED), que emiten dichos haces de luz a diferentes longitudes de onda. Uno de los LED emite una longitud de onda de 940 nanómetros (luz roja) y el otro emite una longitud de onda de 640 nanómetros (luz infrarroja). Una porción de tejido debe ingresar al sensor (normalmente, el dedo índice de la mano) y ser expuesto a los dos frentes de luz. Los elementos como la piel, los huesos, la grasa, entre otros, presentan coeficientes de absorción[49] constantes a la luz. Solo la sangre arterial presenta un nivel de absorción variable (dependiendo de la concentración de oxígeno), lo cual permite que los niveles de luz que atraviesan el dedo sean también variables. Conectando un

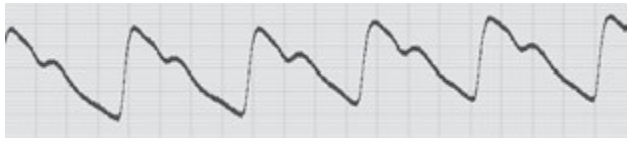
47 Forma de onda de luz variable en el tiempo que se obtiene al medirse en un extremo de un tejido irrigado por sangre arterial que ha sido estimulado en el otro extremo por una fuente luminosa.

48 Equipo comercial de monitoreo médico que permite obtener la información de oximetría de pulso.

49 Toda materia presenta la capacidad de absorber y traslucir la luz. Mientras mayor sea su capacidad de absorber, mayor será su nivel de absorción.

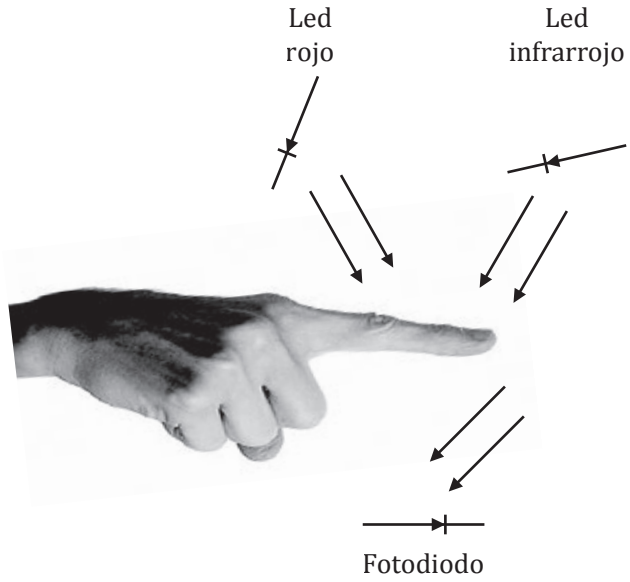
fotodiodo en el lado de la yema del dedo es posible captar estos niveles de luz. La hemoglobina oxigenada presenta un nivel de absorción mayor para la luz infrarroja que para la luz roja, y la hemoglobina no oxigenada presenta niveles de absorción opuestos. Esta relación entre las magnitudes de luz que logran atravesar el tejido permite obtener el parámetro de nivel de saturación de oxígeno. En el gráfico 11.28 se observa el esquema de un clip de oximetría de pulso para dedo.

Gráfico 11.27. Forma de onda típica de una señal fotopletismográfica



Elaboración propia.

Gráfico 11.28. Esquema del dedo índice dentro del clip de oximetría de pulso



Elaboración propia.

El cálculo del nivel de saturación de oxígeno requiere de un algoritmo y de pruebas empíricas con diversos pacientes. El resultado del procesamiento de la información de oximetría de pulso da un número o coeficiente ROS (del inglés *Ratio of Ratios*), el cual se debe colocar en el eje de las abscisas de un plano cartesiano para, según una curva, evaluar el valor del nivel de saturación de oxígeno, el cual se ubica en el eje de las ordenadas. Esta curva se debe obtener en base a pruebas comparativas con equipos comerciales homologados y calibrados que miden el nivel de saturación de oxígeno, y con diversos pacientes que presenten diferentes valores de este parámetro. Las pruebas comparativas permiten ajustar la curva, de manera que los resultados del algoritmo propuesto sean similares al de un equipo comercial[50].

50 Cfr. Webster 1997.

Por otro lado, el pulso cardiaco es un parámetro mucho más simple de obtener y calibrar, ya que puede ser comparado directamente con el pulso del paciente tomado en su muñeca de forma manual. El algoritmo de cálculo del pulso cardiaco es más simple y no requiere de mucho tratamiento matemático.

Finalmente, la onda fotopletismográfica se debe mostrar en una pantalla gráfica. La forma de la onda es muy variable, debido a parámetros como tipo de piel del paciente, grosor del dedo, interferencia electromagnética de línea de 60 Hz y movimiento de la mano del paciente.

El sensor NONIN es un clip de dimensiones 3.2 x 2 x 1.25 centímetros que posee un cable de conexión de 1.5 metros. El dispositivo está especificado para obtener el nivel de saturación de oxígeno y el pulso cardiaco en pacientes adultos con un peso superior a 30 kg y un grosor tisular entre 5 y 21 milímetros. El dedo del paciente (generalmente el dedo índice, medio o anular) se debe introducir con la punta del dedo llegando al tope de la base del clip. El personal a cargo de la monitorización del paciente debe cerciorarse de que la uña de este no sea demasiado larga como para impedir la correcta ubicación del dedo en el clip. Además, el paciente no deberá presentar ningún tipo de esmalte sobre la uña, ya que esto puede impedir el correcto ingreso de luz al fotodiodo. Ante estas situaciones, el personal de enfermería o auxiliar debe proceder con el recorte de la uña y la remoción del esmalte, haciendo uso de acetona. En el gráfico 11.29 se puede observar una imagen de este dispositivo.

Gráfico 11.29. Sensor de oximetría de pulso NONIN modelo IPOD 3212

Fuente: http://www.nonin.com/iPod

En el gráfico 11.30 se muestran las especificaciones técnicas del sensor NONIN. Se aprecia el rango de precisión del terminal y, a su vez, se observan sus rangos de operación ambientales.

Gráfico 11.30. Especificaciones técnicas del IPOD 3212

Rango de medición del nivel de saturación de oxígeno	0 a 100%
Rango de medición del pulso cardiaco	18 a 300 pulsos por minuto
Longitudes de onda medidas	Rojo – 660 nanómetros Infrarrojo – 925 nanómetros

Precisión SpO_2 (70-100%)(±1 SD)[51]	**Sin movimiento** -Adultos ± 2 dígitos **Movimiento** -Adultos ± 3 dígitos **Baja perfusión** -Adultos ± 3 dígitos
Pulso cardiaco	**Sin movimiento (18 a 300 BPM)** -Adultos ± 3 dígitos **Movimiento (40 a 240 BPM)** -Adultos ± 5 dígitos **Baja perfusión (40 a 240 BPM)** -Adultos ± 3 dígitos
Temperatura: a. Operando b. No operando	0ºC a + 50ºC -30ºC a + 50ºC
Humedad Operando No operando	10 a 90% 10 a 95%
Potencia de consumo	60mW
Entrada de voltaje	2 a 6 voltios DC
Salida digital	0 a 5 voltios
Aislamiento de paciente	IEC 60601-1
Corriente de fuga	No se aplica
Peso	100 gramos(incluyendo el cable y conector)
Exposición mecánica Golpe Vibración	IEC 68-2-27 Estándar 810C, método 514-2

Fuente: Manual de operación del IPOD NONIN 3212.

Además, el ipod presenta cinco cables para su conexión a la tarjeta impresa. Cada cable posee una función específica:

Entradas:

Cable rojo = Alimentación (2 a 6 VDC).

Cable negro = Tierra.

Cable de apantallamiento = Tierra.

Salidas:

Cable verde = Salida serial (TTL).

Cable azul = Salida serial (RS232).

51 La desviación estándar es una medida estadística: hasta el 32% de las lecturas pueden fallar fuera de estos límites.

De igual modo, es muy importante seguir ciertas recomendaciones en el uso del ipod 3212, para su correcto funcionamiento y el cuidado del paciente:

1. Para poder garantizar la alineación correcta del sensor y la integridad de la piel del dedo, se debe inspeccionar el sitio de aplicación del sensor cada 4 horas como mínimo. La sensibilidad del paciente puede variar según su condición clínica o sensibilidad de la piel.
2. El cable del paciente se debe colocar adecuadamente para evitar la posibilidad de enredos o estrangulación.
3. Si el ipod se encuentra dañado, no debe ser utilizado bajo ninguna circunstancia.
4. El dispositivo no se debe hacer funcionar sobre ningún otro equipo. Si es necesario, hacerlo trabajar en forma adyacente a otro terminal de monitoreo. Se debe observar continuamente el dispositivo para verificar que funcione correctamente.

El ipod 3212 posee un controlador que se encarga de procesar información del nivel de saturación de oxígeno, pulso cardiaco, muestras de la onda fotopletismográfica, estado del sensor, sincronismo y corrección de errores. Los datos se transmiten en un formato serial a 9600 baudios sin bit de paridad, 8 bits de datos y 1 bit de parada. La tasa de transferencia de datos transmite 75 paquetes de datos cada segundo. Cada paquete de datos contiene 5 bytes.

En el gráfico 11.31 se muestra un ejemplo tomado de un paciente adulto sano. En esta tabla se observa una trama de datos completa que involucra 25 paquetes de datos.

Gráfico 11.31. Ejemplo de una trama completa de datos enviados por el sensor

BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	Contador de paquete
1	129	57	0	187	0
1	128	59	90	22	1
1	128	61	97	31	2
1	128	64	14	207	3
1	128	67	0	196	4
1	128	73	0	202	5
1	128	82	0	211	6
1	128	96	0	225	7
1	128	115	0	244	8
1	130	137	97	109	9
1	130	160	97	132	10
1	130	178	0	53	11
1	130	190	0	65	12
1	130	194	0	69	13
1	130	191	90	156	14
1	130	183	97	155	15
1	130	172	0	47	16
1	130	160	0	35	17

BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	Contador de paquete
1	130	147	0	22	18
1	128	135	0	8	19
1	128	124	0	253	20
1	128	116	0	245	21
1	128	110	0	239	22
1	128	107	0	236	23
1	128	106	0	235	24

Fuente: Manual de operación del IPOD NONIN 3212.

En el gráfico 11.31 se puede observar que una trama completa contiene 25 paquetes de datos, y que cada paquete está compuesto de 5 bytes. En total se transmiten tres tramas cada segundo. Existe una variable muy importante: el contador de paquetes. Esta variable se incrementa cada vez que el ipod transmite 5 bytes. Al llegar al valor de 24 (25 paquetes enviados), esta variable debe inicializarse en cero para iniciar la cuenta de una nueva trama.

Cada paquete compuesto por 5 bytes siempre comienza con el primer byte (BYTE 1) tomando el valor de 00000001_2. Este valor sirve como sincronismo y siempre indica el inicio de un nuevo paquete. El segundo byte (BYTE 2) corresponde al estado del ipod. Algunas veces el paciente presenta ciertas características que hacen que el ipod no pueda obtener los parámetros de medición de forma correcta. Para esto, el valor que puede tomar este byte siempre variará de 10000000_2 a 11111111_2, y el significado de cada bit indicará una determinada circunstancia adversa, como se muestra en el gráfico 11.32.

Gráfico 11.31. Bits del byte de estado (BYTE 2)

BIT 0	Sincronismo de trama. Es '1' cuando el contador de paquetes toma el valor de '0'.
BIT 1	Perfusión verde (Flujo sanguíneo en buen nivel). Es '1' cuando la condición es verdadera.
BIT 2	Perfusión roja (Flujo sanguíneo muy bajo). Es '1' cuando la condición es verdadera.
BIT 3	Alarma del Sensor. Es '1' cuando la condición es verdadera.
BIT 4	Fuera de rango. Es '1' cuando la condición es verdadera.
BIT 5	Mal pulso. Es '1' cuando la condición es verdadera.
BIT 6	Sensor desconectado. Es '1' cuando la condición es verdadera.
BIT 7	Siempre es '1'.

Fuente: Manual de operación del IPOD NONIN 3212.

El tercer byte representa las muestras de la onda fotopletismográfica. Las muestras poseen una frecuencia de muestreo de 75 Hz, ya que cada paquete siempre presenta un valor válido en el BYTE 3. Si se graficaran las muestras contenidas en el BYTE 3 en una pantalla GLCD o una pantalla de TV, la apariencia de la gráfica sería como la del gráfico 11.27.

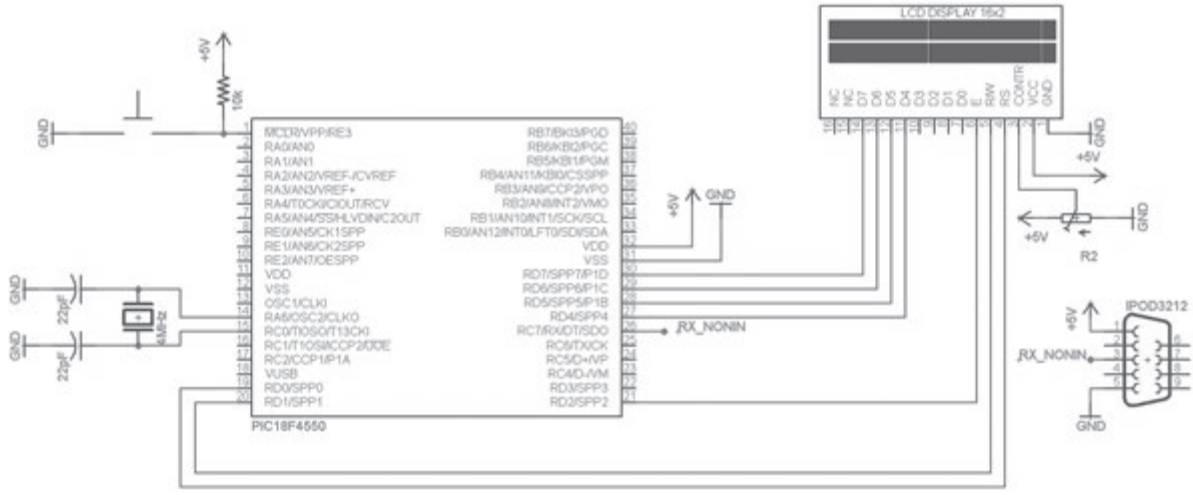
El cuarto byte (BYTE 4) posee información tanto del nivel de saturación de oxígeno como del pulso cardiaco y de la versión del *firmware* del controlador del ipod. La información depende directamente del contador de paquete (*Frame count*). Si el contador de paquete se encuentra en el valor 2, 8, 9, 10, 15, 16 o 22, entonces el BYTE 4 poseerá información del nivel de saturación de oxígeno. Por otro lado, si el contador de paquete se encuentra en el valor 1, 13, 14, 19 o 20, la información contenida en el BYTE 4 representará el pulso cardiaco. En el caso exclusivo de que el contador de paquete se encontrara en el valor 3, el contenido del BYTE 4 será la versión del *firmware*. Es posible utilizar este valor para sincronizar el contador de paquete al momento de implementar el algoritmo de decodificación de los bytes enviados.

Finalmente, el quinto BYTE representa la información de verificación de error (*checksum*). El BYTE 5 siempre debe ser el resultado de la suma del contenido del BYTE 1, el BYTE 2, el BYTE 3 y el BYTE 4. En el gráfico 11.31, el contenido del BYTE 5 siempre resulta ser la suma de los 4 BYTES anteriores. En caso de que la suma supere el valor de 255, el resultado será el valor de desbordamiento de 8 bits. De no ser así, alguno de los 4 BYTES anteriores presentarán un error y el microcontrolador que se encarga de decodificar los datos deberá obviar ese paquete.

En el gráfico 11.33 se observa el circuito propuesto para monitorear el nivel de saturación de oxígeno y el pulso cardiaco de un paciente. En la pantalla LCD se mostrarán estos valores con tres dígitos. El pulso es una variable que puede alternar entre 30 bpm y 300 bpm, en el peor de los casos; mientras que el nivel de saturación de oxígeno puede alternar entre 0 y 100%. Los valores normales para un paciente estable se encuentran entre 60 bpm y 100 bpm (en condición de reposo) y entre 93% y 100% de SPO_2.

El cable terminal del sensor ipod 3212 se ha soldado a un conector DB9 aéreo, por lo cual se representa este conector en el circuito esquemático del gráfico 11.33 como el sensor. La señal de salida del sensor ingresa directamente al pin RC7 de recepción de EUSART del PIC18F4550, con lo cual solo se utilizará el evento de recepción serial para detectar los parámetros del paciente. Los valores de saturación de oxígeno y pulso cardiaco se mostrarán en la pantalla LCD siempre y cuando no ocurran tres eventos: el paciente no haya conectado el dedo dentro del sensor, el sensor no pueda detectar el nivel de saturación de oxígeno o el sensor no pueda detectar el pulso. Estas condiciones pueden ocurrir si el dedo del paciente presenta un bajo flujo sanguíneo, si la pigmentación de la piel es muy oscura o si se ha aplicado algún tipo de esmalte sobre la uña.

Gráfico 11.33. Circuito esquemático para un prototipo de oxímetro de pulso usando la pantalla LCD



Elaboración propia.

Para poder implementar el programa en el microcontrolador en lenguaje ANSI C, es necesario el uso de una nueva librería de nombre *Serial*. Esta librería posee tres funciones: una para inicializar el módulo EUSART a una velocidad definida por el usuario, otra para transmitir un carácter, y otra para transmitir toda una cadena de caracteres (similar a la usada para generar mensajes completos en la pantalla LCD). El proceso de recepción de datos se maneja en la interrupción por el evento de recepción del EUSART. La función *Abrir_Serial()* se encarga de activar dicha interrupción. Esto obliga a crear la rutina de interrupción en la rutina principal. En el programa 11.2 se muestra el contenido del archivo Serial.h, mientras que en el programa 11.3 se ofrece el contenido del archivo Serial.c que corresponde a la librería Serial.

Programa 11.2. Librería Serial.h

```
#define _115200  25
#define _57600   51
#define _38400   77
#define _19200   155
#define _9600    311
#define _4800    624
void TX_MENSAJE_EUSART(const unsigned char *vector,unsigned char pos);
void TX_CHAR_EUSART(char num);
void Abrir_Serial(int velocidad);
```

Programa 11.3. Librería Serial.c

```
#include <xc.h>
#include "Serial.h"
void TX_CHAR_EUSART(char num)
{
        TXREG = num;
        while(TXSTAbits.TRMT == 0);
}
void Abrir_Serial(int velocidad)
{
        SPBRG = (unsigned char)velocidad;
        SPBRGH = velocidad>>8;
        TXSTAbits.SYNC = 0;             //Modo asíncrono
        TXSTAbits.BRGH = 0;
        BAUDCONbits.BRG16 = 1;
        RCSTAbits.SPEN = 1;             //Habilitar módulo EUSART
        TXSTAbits.TXEN = 1;             //Habilitar la transmisión
        RCSTAbits.CREN = 1;
        PIE1bits.RCIE = 1;              //Habilitar la interrupción por rx
        INTCONbits.PEIE = 1;
        INTCONbits.GIE = 1;
}
void TX_MENSAJE_EUSART(const unsigned char *vector,unsigned char pos)
{
        unsigned char i = 0;
        for(i = 0; i<pos; i++)
        {
                TX_CHAR_EUSART(vector[i]);
        }
}
```

En el programa 11.4 se muestra el algoritmo que realiza la lectura de datos del sensor NONIN. Este programa realiza todo el proceso de análisis de los parámetros del pulso, nivel de saturación de oxígeno y estado del sensor dentro de la rutina de interrupción por recepción serial. El sincronismo de trama se realiza mediante la variable frame_count, que se sincroniza con el código de identificación del sensor, cuyo valor es 0x29 y corresponde a un valor de frame_count de 3. Una vez sincronizado este valor, es posible extraer el pulso y el nivel de saturación de oxígeno para los valores 1 2, y 3, respectivamente.

Programa 11.4. Algoritmo de detección del pulso cardiaco y del nivel de saturación de oxígeno

```
#include <xc.h>
#pragma config FOSC   = XTPLL_XT        //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV = 1               //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV = OSC1_PLL2   //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON
#include "Serial.h"
#include "LCD.h"
#define  VERDADERO  1
#define  FALSO      0
char bandera_NO_SE_DETECTA_SPO2 = 0;
char bandera_MAL_PULSO    = 0;
char bandera_DESCONECTADO = 0;
unsigned int pulso;
char SPO2, bandera_LISTO=FALSO;
void main()
{
        TRISD = 0x08;
        TRISC = 0b10111111;
        Delay10KTCYx(12);
        LCD_CONFIG();
        Abrir_Serial(_9600);
        while(1)
        {
                if(bandera_LISTO == VERDADERO)
                {
                        if(bandera_NO_SE_DETECTA_SPO2 == VERDADERO)
                        {
                                BORRAR_LCD();
                                ESCRIBE_MENSAJE("No se detecta el",16);
                                POS_CURSOR(2,0);
                                ESCRIBE_MENSAJE("SPO2",4);
                        }
```

```
                        else if(bandera_MAL_PULSO == VERDADERO)
                        {
                                BORRAR_LCD();
                                ESCRIBE_MENSAJE("No se detecta el",16);
                                POS_CURSOR(2,0);
                                ESCRIBE_MENSAJE("pulso",5);
                        }
                        else if(bandera_DESCONECTADO  == VERDADERO)
                        {
                                BORRAR_LCD();
                                ESCRIBE_MENSAJE("Sensor",6);
                                POS_CURSOR(2,0);
                                ESCRIBE_MENSAJE("desconectado",12);
                        }
                        else
                        {
                                BORRAR_LCD();
                                ESCRIBE_MENSAJE("Pulso:",6);
                                POS_CURSOR(2,0);
                                ESCRIBE_MENSAJE("SPO2: ",6);
                                POS_CURSOR(1,7);
                                ENVIA_CHAR(pulso/100+'0');
                                ENVIA_CHAR((pulso%100)/10+'0');
                                ENVIA_CHAR((pulso%100)%10+'0');
                                ESCRIBE_MENSAJE("bpm",3);
                                POS_CURSOR(2,7);
                                ENVIA_CHAR(SPO2/100+'0');
                                ENVIA_CHAR((SPO2%100)/10+'0');
                                ENVIA_CHAR((SPO2%100)%10+'0');
                                ENVIA_CHAR('%');
                                bandera_LISTO = FALSO;
                        }
                }
        }
}
void interrupt RUT_INT(void)
{
        static char frame_count,i = 0;
        static char BYTE2,BYTE3,BYTE4,BYTE5;
        char dato;
        if(PIR1bits.RCIF==1)
```

```
        {
                dato = RCREG;
                if(dato == 1) {i = 0; frame_count++;}
                i++;
                if(i==2) BYTE2 = dato;
                if(i==3) BYTE3 = dato;
                if(i==4) BYTE4 = dato;
                if(i==5) BYTE5 = dato;

                if(BYTE2&0x10)bandera_NO_SE_DETECTA_SPO2= VERDADERO;
                else          bandera_NO_SE_DETECTA_SPO2 = FALSO;
                if(BYTE2 & 0x20) bandera_MAL_PULSO       = VERDADERO;
                else          bandera_MAL_PULSO       = FALSO;
                if(BYTE2 & 0x40) bandera_DESCONECTADO    = VERDADERO;
                else          bandera_DESCONECTADO    = FALSO;
                if(BYTE4==41)        frame_count = 3;
                if(frame_count == 24) frame_count = 0;
                if(frame_count == 0)  {pulso = BYTE4;pulso<<8;}
                if(frame_count == 1)  {pulso+= BYTE4;}
                if(frame_count==2){SPO2=BYTE4;bandera_LISTO = VERDADERO;}

                PIR1bits.RCIF=0;
        }
}
```

Video

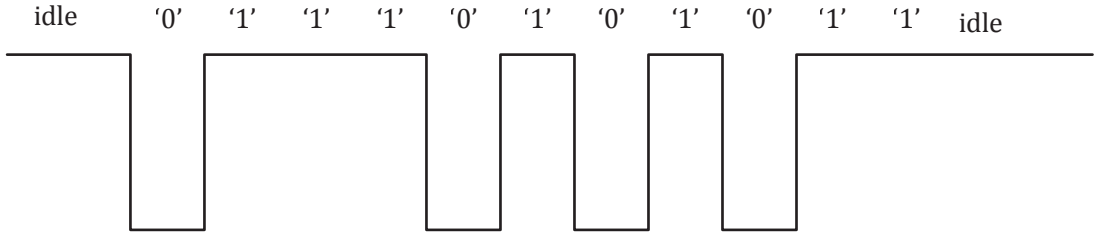
Diseño de un voltímetro digital con transmisión de los datos a un computador a través del puerto serie.

http://bit.ly/1UBAMXq

Resolución de videos en HD

Preguntas de repaso

1. Dibuje una trama serial asíncrona en formato NRZ que codifique el byte 0xA2 usando un bit de paridad Par.
2. Una trama serial asíncrona sin bit de paridad y en formato NRZ tiene un tiempo de duración de 520 µsegundos. ¿A qué tasa de bits se está transmitiendo esta trama?
3. ¿Cuál es la función del circuito integrado MAX232 en la comunicación RS232 entre el PIC18F4550 y el computador?
4. ¿Cómo se determina la probabilidad de error de transmisión serial asíncrona en el PIC18F4550?
5. Analice la trama serial TTL asíncrona con bit de paridad Par enviada desde el pin RC6/TX del PIC18F4550 a una de las entradas del circuito integrado MAX232 a una velocidad de 9600 baudios.



   a. ¿Cuál es el tiempo de duración de toda la trama compuesta por 11 bits?
   b. Dibuje la señal de salida del MAX232. Indique los valores máximo y mínimo de la señal y cuáles valores corresponden a un '0' o '1' lógico.
   c. Indique si la trama presenta error de paridad. Justifique su respuesta.

Respuestas

Respuestas del capítulo 11:

http://bit.ly/1UyXzTM

Capítulo 12. La interfaz serial periférica

La interfaz serial periférica SPI (de las siglas en inglés *Serial Peripheral Interface*) es un protocolo de comunicación serial síncrono creado por la compañía Motorola que se ha convertido en un estándar de comunicación en la gran mayoría de sistemas embebidos. El protocolo SPI permite la comunicación de un sistema embebido con uno o varios dispositivos periféricos a través de un bus de comunicación compartido. Cada año aparecen nuevos dispositivos periféricos con este protocolo de comunicación. Entre los más comunes se tienen acelerómetros, memorias EEPROM, relojes en tiempo real, memorias SD[52], conversores analógicos digitales y otros más. Por esta razón, conocer la correcta configuración del módulo SPI de un sistema embebido permite integrar una gran cantidad de dispositivos externos a un proyecto de diseño digital.

A continuación se explican los fundamentos del protocolo SPI, el formato de la señal de información y la configuración del módulo MSSP del PIC18F4550, y se brindan algunos ejemplos de aplicación con dispositivos periféricos comerciales.

12.1 Fundamentos de la comunicación serial síncrona SPI

La comunicación SPI se basa en el uso de tres líneas de comunicación para la transferencia de paquetes de 8 bits de información desde un dispositivo maestro hacia otro u otros dispositivos esclavos. La comunicación entre ambos dispositivos es bidireccional y Full dúplex, lo que implica que la transmisión y recepción se puede realizar en simultáneo.

Algunos dispositivos periféricos SPI (que cumplen la función de esclavos) solamente se encargan de transmitir información, mas no de recibirla (como puede ser el caso de un acelerómetro de medición de un solo eje). Otros solo reciben información, mas no la transmiten (como es el caso de algunos convertidores digitales analógicos) y otros pueden realizar ambas operaciones (como sería el caso de una memoria EEPROM).

El maestro en una comunicación SPI normalmente es un sistema embebido. La función de este en la comunicación es la de generar la señal de reloj y los comandos necesarios para que el o los dispositivos esclavos se puedan configurar. Además, el maestro es el encargado de seleccionar cuál de los dispositivos esclavos deberá estar activo para la comunicación.

En el esquema del gráfico 12.1, el controlador se conecta a tres dispositivos esclavos: Esclavo 1, Esclavo 2 y Esclavo 3. Los tres dispositivos comparten el mismo bus de comunicaciones. Este bus se divide en tres líneas: SDO, SCK y SDI. Las dos primeras transfieren la información del maestro a los esclavos y la última del esclavo al maestro. Para evitar la colisión de la información el maestro maneja tres líneas independientes: CS1, CS2 y CS3 (CS viene de las siglas en inglés *Chip Select* o selector de chip). Estas líneas habilitan el funcionamiento de cada esclavo cuando se le aplica un nivel '0'. Cuando esta línea se configura en un nivel '1', el dispositivo esclavo se deshabilita colocando sus líneas del bus

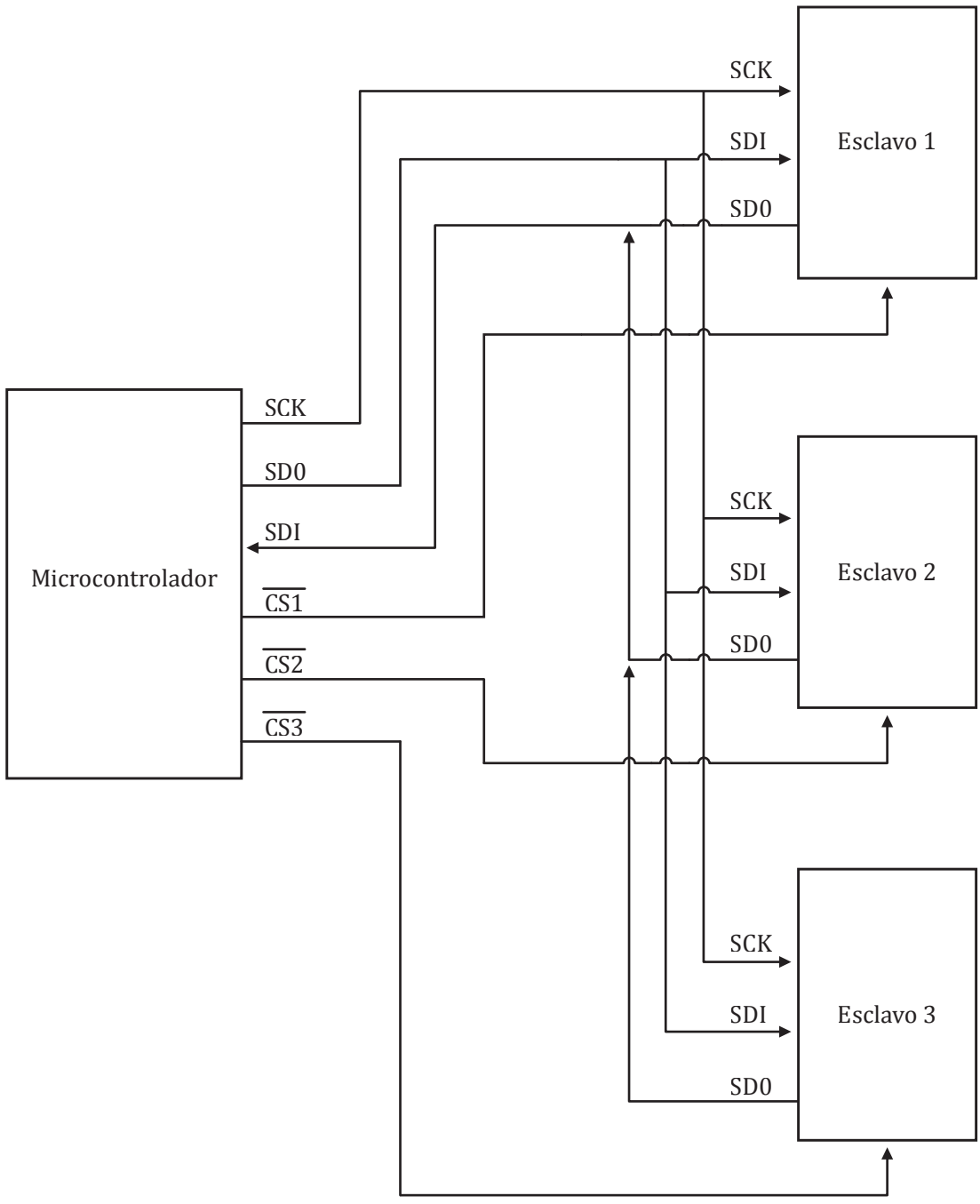
52 *Secure Digital.*

de comunicación en alta impedancia. El maestro debe tener mucho cuidado de solo habilitar un dispositivo esclavo a la vez para evitar posibles colisiones en las líneas de comunicación.

12.2 Pines del SPI

Los pines del SPI son cuatro: SCK, SDO, SDI y CS. Como se puede observar en el gráfico 12.1, tanto el maestro como los esclavos poseen las mismas líneas de comunicación, pero la función es diferente en cada caso.

Gráfico 12.1. Esquema de la comunicación SPI entre el maestro y los esclavos



Elaboración propia.

La línea SCK es la encargada de generar la señal de reloj de sincronismo para el caso del maestro. La misma línea SCK es una entrada en el esclavo y sirve para recibir la señal de reloj, de forma tal que se detecten los flancos activos para realizar la lectura y generación de los datos. Las líneas SCK del maestro y del esclavo deben ser conectadas de forma directa, operando como salida para el primero y entrada para el segundo, respectivamente. La señal de reloj solo genera 8 pulsos o periodos por cada byte de transmisión y la frecuencia de los pulsos es configurable. Muchos dispositivos esclavos presentan limitaciones en el valor máximo de frecuencia de reloj. Por esta razón, el maestro debe estar en la capacidad de reducir su velocidad de reloj para adecuarse a los requerimientos del dispositivo esclavo. Normalmente, la frecuencia de reloj puede variar entre unos cientos de Hz hasta decenas de MHz.

La línea SDO (del inglés *Serial Data Output*), también conocida como MOSI (Salida de maestro y entrada de esclavo, del inglés *Master Out Slave In*), es por la cual el maestro envía los 8 bits de datos en sincronía con la señal de reloj. Cada bit aparece en la línea sincronizado con uno de los flancos activos de la señal de reloj. Por defecto, esta línea permanece en '1' cuando no hay actividad en el bus. Esta línea debe estar conectada directamente con el pin SDI (del inglés *Serial Data Input*) del esclavo, ya que los datos de salida deberán ingresar a este dispositivo para ser recibidos y procesados.

Por otro lado, la línea SDI, también conocida como MISO (Entrada del maestro y salida del esclavo, del inglés *Master In Slave Out*) permite muestrear los 8 bits de datos enviados por el pin SDO del esclavo. Durante la transmisión de un byte por la línea SDO es posible recibir en simultáneo la respuesta del esclavo por el pin SDI. Por esta razón, la comunicación es considerada Full dúplex.

Finalmente, la línea CS permite seleccionar el esclavo que estará disponible para la comunicación. El maestro puede manejar múltiples líneas CS para poder operar a diversos esclavos en la red. Se recomienda que este número de esclavos no sea mayor que 8. En muchos sistemas embebidos (como es el caso del PIC18F4550) las líneas CS no son parte del módulo SPI, sino que son pines de E/S convencionales cuya función se elige para este fin.

12.3 Formatos de transmisión del bus SPI

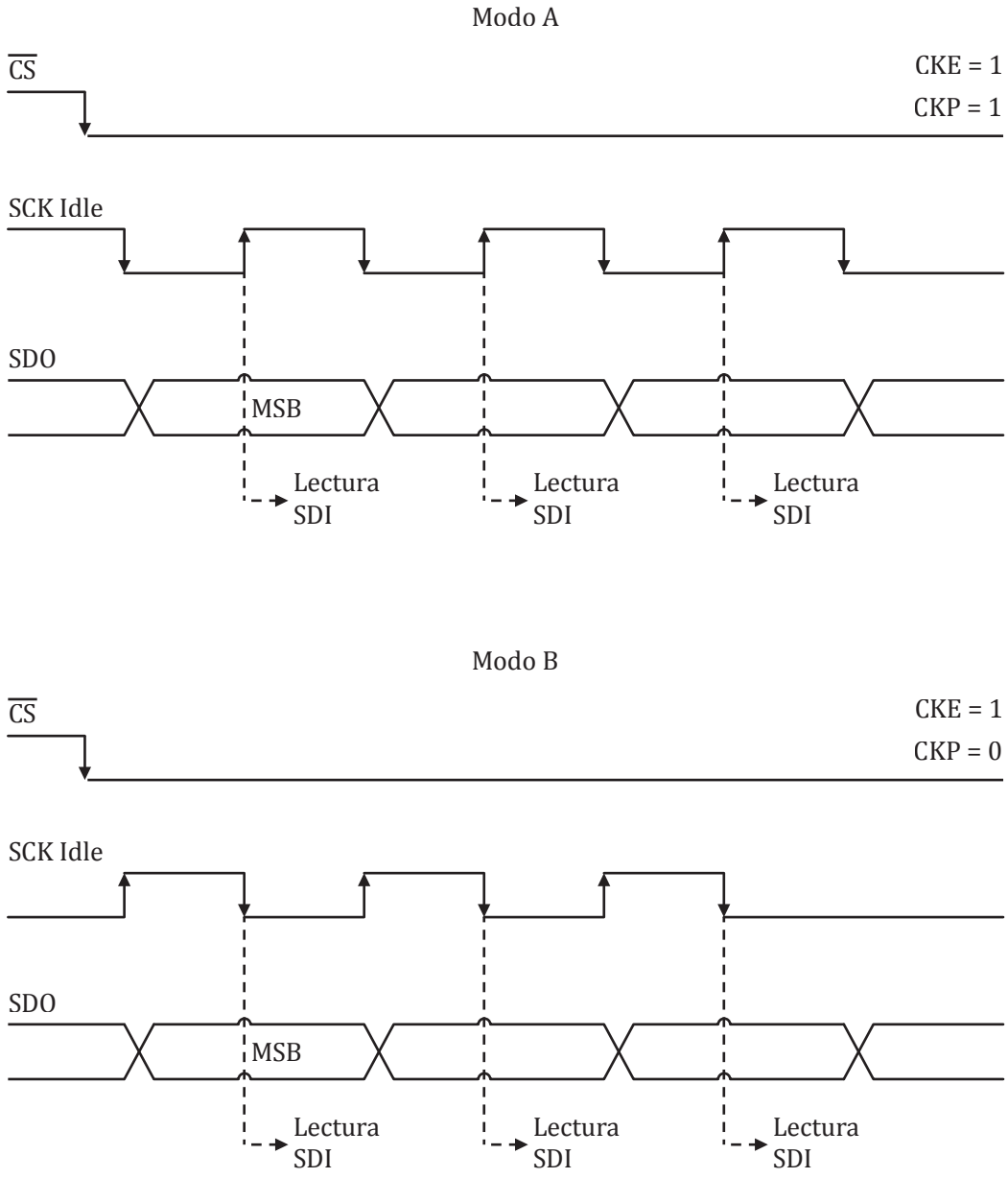
Es muy importante notar que el sincronismo entre las líneas SDO, SDI y SCK puede variar de acuerdo con las características técnicas del protocolo SPI del dispositivo esclavo. Por esta razón, la mayoría de módulos SPI maestros de sistemas embebidos permiten configurar tanto la polaridad de la señal de reloj como la fase de escritura en SDO y la de lectura de la línea SDI.

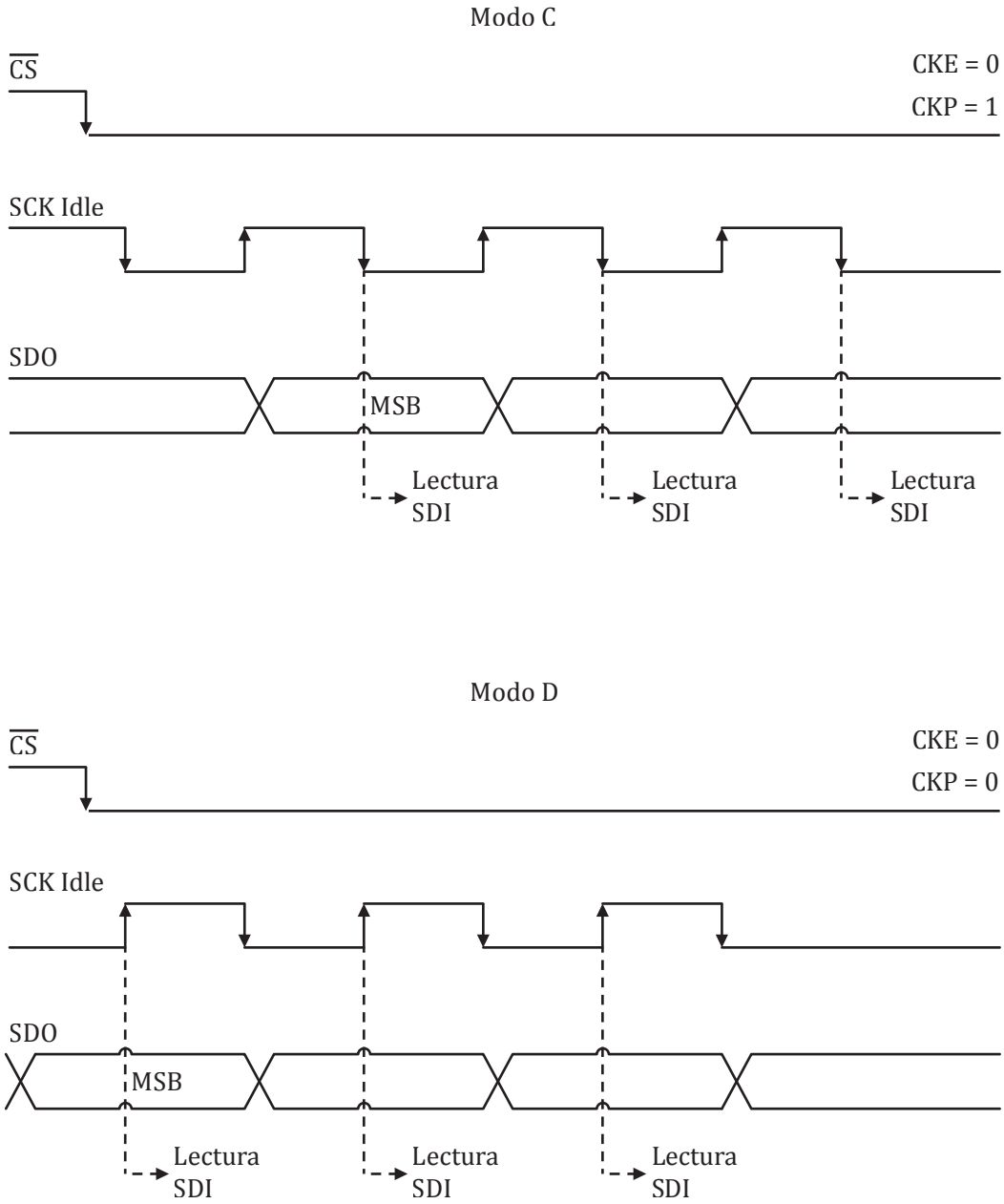
Para configurar tanto la polaridad como fase se cuenta con 2 bits de configuración denominados CKP y CKE. El primer bit configura la polaridad de la señal de reloj; esto significa si el estado de inactividad o *Idle* de la señal es un nivel alto (CKP = '1') o un nivel bajo (CKP = '0'). El segundo bit configura la fase, lo cual determina el flanco de reloj activo para el desplazamiento del dato a transmitir en la línea SDO. De acuerdo con la combinación de estos dos bits, existen cuatro modos de configuración de la polaridad y la fase, conocidos como los modos A, B, C y D, los cuales se presentan en el gráfico 12.2. Por ejemplo, en este gráfico se puede observar que en el modo A la polaridad de reloj del maestro es alta (estado *Idle* en '1'), por lo cual el primer flanco activo de la señal de reloj será el de bajada. En el bus SPI el bit MSB se transmite primero; por tal motivo, este bit será puesto en la línea SDO en sincronismo con el primer flanco activo de reloj o primer flanco de bajada. En el modo B se puede observar que la polaridad de reloj es baja (estado *Idle* en '0'), por lo que el primer flanco activo será el de subida. Es así que en este modo el bit a actualizar en la línea SDO estará en sincronía con los flancos de subida del

reloj. Para los modos C y D, el cambio que se puede observar se encuentra en la fase. El bit a transmitir en la línea SDO se sincroniza con el segundo flanco activo de la señal de reloj, lo cual implica que el dato se actualizará medio periodo después del primer flanco.

Uno de los aspectos que se debe observar es que la lectura de la señal SDI del maestro se realiza en el segundo flanco activo (medio periodo) o en el tercer flanco activo (un periodo) de la señal de reloj. Este proceso no se puede realizar en sincronía con el primer flanco activo, ya que este evento es requerido por el dispositivo esclavo para actualizar el dato en la línea SDI. La configuración de la fase de lectura de la línea SDI es independiente del modo.

Gráfico 12.2. Formatos de fase y polaridad de las señales del bus SPI





Elaboración propia.

12.4 El módulo SPI del PIC18F4550

El PIC18F4550 cuenta con un módulo de nombre MSSP (del inglés *Master Synchronous Serial Port*) que permite implementar tanto el protocolo SPI como el I^2C. Este módulo utiliza el pin RC7 como SDO, el pin RB0 como SDI y el pin RB1 como SCK. Además, el pin RA5 puede actuar como CS, pero solo cuando el módulo MSSP se configura en modo esclavo (para ser controlado por otro sistema embebido).

La configuración de los registros TRISC y TRISB es fundamental en la operación del bus SPI. Es necesario que el bit 7 del registro TRISC se configure como salida y los bits 1 y 0 del registro TRISB sean configurados como salida y entrada digital, respectivamente.

El funcionamiento del módulo SPI se basa en un registro de desplazamiento de nombre SSPSR, el cual desplaza tanto los 8 bits hacia al pin SDO como los bits de entrada provenientes del pin SDI hacia este registro. SSPSR trabaja directamente con el registro SSPBUF, el cual es un doble buffer de lectura y escritura. Esto significa que, para un proceso de escritura, el contenido de SSPBUF se cargará en SSPSR y generará un proceso de transmisión en SDO. Por otra parte, una lectura del registro SSPBUF permite obtener el byte recibido proveniente del dispositivo esclavo. En el gráfico 12.3 se puede observar el diagrama de bloques del módulo MSSP.

La señal de reloj generada por el módulo SPI depende directamente de la frecuencia de la señal del CPU F_{OSC}, o también del valor de comparación del módulo TMR2. Para el primer caso, se cuenta con un Prescaler programable con divisores de 4, 16 y 64, los cuales permiten reducir la frecuencia del orden de los MHz hasta los kHz. Con un F_{OSC} = 48MHz, se puede tener una tasa máxima de 12 Mbps.

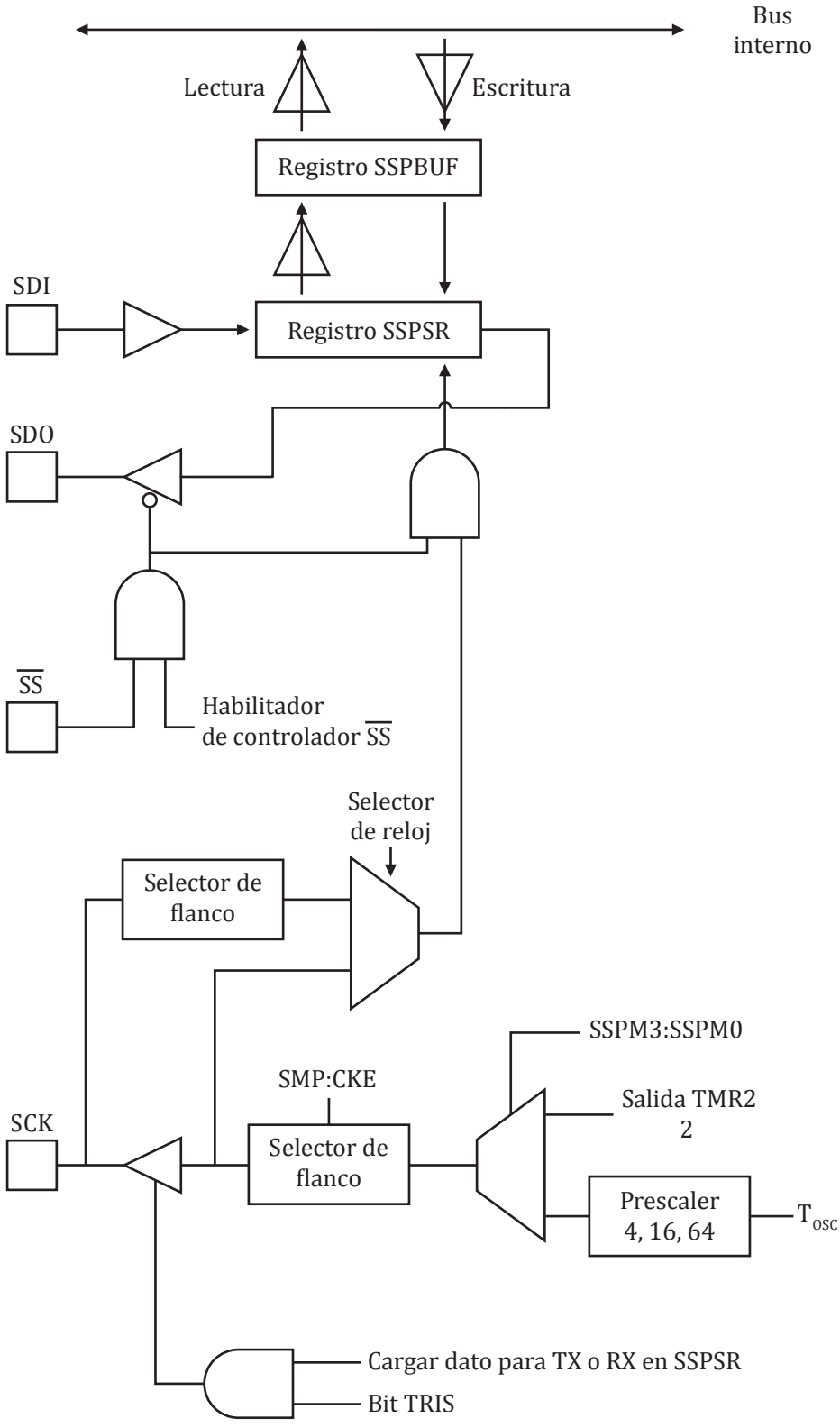
Adicionalmente, la salida de reloj posee un selector de flanco activo, fase de generación del bit de transmisión en la línea SDO y fase de lectura del bit en la línea SDI. El pin SCK del PIC18F4550 puede operar como salida (modo maestro) y como entrada (modo esclavo). Es importante que al ser configurado como esclavo la configuración de la señal de reloj pertenezca a un modo compatible con el maestro.

La señal de reloj SCK, entonces, estimula al registro SSPSR a desplazarse en dos sentidos opuestos: uno para generar el dato en SDO y otro para recibir el byte proveniente de SDI. La frecuencia y el flanco activo seleccionado de SCK definen el momento de cada desplazamiento. Una vez que el registro SSPSR se ha desplazado 8 posiciones, el bit de bandera SSPIF se pone a nivel '1', lo cual también puede ser usado para activar una interrupción.

Cada vez que se realiza una transmisión por el pin SDO, siempre se efectuará una lectura obligatoria del pin SDI. Esto es un proceso automático. ¿Qué sucede, entonces, si el dispositivo esclavo no presenta información para enviar al maestro? ¿Qué valor se deberá cargar en el registro SSPBUF? Pues en el protocolo SPI la inactividad de la línea se representa con el valor 0xFF. Cualquier lectura de SSPBUF que contenga este valor implica que no se ha recibido dato alguno del dispositivo esclavo. De la misma manera, si solo se desea realizar una lectura del dispositivo esclavo, mas no enviar ningún byte de datos o configuración a través de SDO, se deberá transmitir el dato 0xFF colocándolo en el registro SSPBUF.

Cuando se opera en modo maestro, todas las transmisiones de paquetes por SDO y SDI deben estar precedidas de un pin CS en nivel '0' o '1' (dependiendo de la configuración del dispositivo esclavo) que habilite al esclavo. El pin CS en modo maestro no actúa de forma automática, por lo cual el programador debe asignar la función de habilitador a cualquier otro pin de E/S y mediante código debe poner el pin a nivel '0' o '1' (según sea el caso) antes de realizar una escritura en el registro SSPBUF.

Gráfico 12.3. Diagrama de bloques del módulo SPI



Fuente: Hoja de datos del PIC18F4550.

12.4.1 Registros de configuración del módulo SPI

Los registros del módulo SPI se dividen en datos y control. Los primeros ya se explicaron, como el registro doble buffer SSPBUF y el registro SSPSR, que no es directamente accesible. Los registros de control son únicamente dos: SSPSTAT y SSPCON1.

En el gráfico 12.4 se muestra el registro SSPSTAT. Los dos bits más significativos son de lectura y escritura, mientras que el resto son solo de lectura. La función de estos bits se detalla a continuación.

Gráfico 12.4. El registro SSPSTAT

SMP	CKE	D/A	P	S	R/W	UA	BF

Elaboración propia.

Bit SMP: este bit permite definir el momento en el cual se realizará el muestreo de la línea de datos SDI cuando el módulo SPI se encuentra configurado en modo maestro. Si SMP es '1', el muestreo se realizará al finalizar el periodo de reloj de la señal. En caso de que el bit sea '0', el muestreo será realizado a la mitad del periodo (tal como aparece indicado en el gráfico 12.2 en los modos C y D). En modo esclavo, este bit debe estar fijado en nivel '0'.

Bit CKE: permite configurar el flanco activo para la generación del bit en SDO. Si CKE es '0', la transmisión ocurrirá en la transición de reloj activo a estado de *Idle* (modos C y D), y, en caso contrario, la transición será de estado de *Idle* a estado activo del reloj (modos A y B).

Los bits D/A, P, S, R/W y UA solo tienen efecto en el modo de operación I^2C y serán explicados en el siguiente capítulo.

Bit BF: es la bandera de recepción completa. Esta se pone a nivel '1' cuando el registro SSPBUF se encuentra lleno y listo para ser leído.

Para complementar las funciones de configuración del módulo SPI se cuenta con el registro SSPCON1, que se muestra en el gráfico 12.5. Este registro, a diferencia del registro SSPSTAT, sí posee sus 8 bits completos dedicados a las funciones del módulo SPI. Todos los bits son de acceso a lectura y escritura. A continuación se explica el funcionamiento de cada uno de sus bits.

Gráfico 12.5. El registro SSPCON1

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0

Elaboración propia.

Bit WCOL: este bit representa el estado de colisión en caso de que se escriba el registro SSPBUF cuando aún el registro SSPSR se encuentra transmitiendo un byte previo. Para evitar esta condición, se debe observar el estado de este bit. Si se encuentra en nivel '1', entonces no es posible realizar una transmisión del nuevo byte.

Bit SSPOV: indica el estado de desbordamiento en la recepción del dato. Esto significa que se ha recibido un byte completo por el pin SDI y este byte ha desplazado el valor anterior almacenado en SSPBUF, el cual no fue previamente leído. Ante esta situación, el bit SSPOV se pone automáticamente a nivel '1', indicando la pérdida del byte anterior.

Bit SSPEN: este bit permite activar al módulo SPI, haciendo que las líneas SCK, SDO y SDI se comporten como pines seriales. Si este bit es '1', el módulo se encontrará activo.

Bit CKP: permite configurar la polaridad del reloj. Si CKP es '1', entonces el estado de *Idle* será un nivel alto, mientras que, si CKP es '0', el estado de *Idle* será bajo.

Bits SSPM3:SSPM0: estos cuatro bits configuran la frecuencia de reloj y el modo de operación del módulo SPI (maestro o esclavo). En el gráfico 12.6 se observan las opciones de configuración que permiten seleccionar estos bits.

Gráfico 12.6. Opciones de configuración de los bits SSPM3:SSPM0

SSPM3	SSPM2	SSPM1	SSPM0	Configuración
0	0	0	0	SPI en modo maestro. La frecuencia de reloj será $F_{osc}/4$.
0	0	0	1	SPI en modo maestro. La frecuencia de reloj será $F_{osc}/16$.
0	0	1	0	SPI en modo maestro. La frecuencia de reloj será $F_{osc}/64$.
0	0	1	1	SPI en modo maestro. La frecuencia de reloj será la salida de TMR2/2.
0	1	0	0	SPI en modo esclavo. La habilitación se obtiene por el pin RA5.
0	1	0	1	SPI en modo esclavo sin habilitación por el pin RA5.

Elaboración propia.

En el programa 12.1 se muestra un ejemplo muy sencillo en el que se espera recibir un dato por el bus SPI configurado en modo maestro. Una vez que se detecta la recepción, el dato se guarda en la memoria SRAM y se transmite un nuevo valor al dispositivo esclavo.

Programa 12.1. Ejemplo de recepción y transmisión del SPI

```
org 0x00
        goto MAIN
org 0x08
        retfie
org 0x018
        retfie
org 0x020
```

```
MAIN:
        bsf  TRISB,0
        bcf  TRISB,1
        bcf  TRISC,7
        clrf SSPSTAT
        movlw 0x22
        movwf SSPCON1
ESPERA:
       btfss   SSPSTAT,BF
       goto   ESPERA
       movf  SSPBUF,W
       movwf 0x20,0
       movf    0x21,W,0
       movwf  SSPBUF
LAZO:
       btfss  PIR1,SSPIF
       goto LAZO
FIN:
      goto FIN
      END
```

12.5 Ejemplos de aplicación con el bus SPI del PIC18F4550

A continuación se ofrecen dos ejemplos de aplicación del bus SPI para el manejo de un reloj en tiempo real externo (como el DS1305) y un controlador de matrices de LED (como el MAX6952). Para ambos ejemplos se hace uso de una librería SPI, la cual permitirá el empleo de tres funciones: una de inicialización del módulo SPI en modo maestro, otra de transmisión de datos y otra de recepción. En el programa 12.2 se muestra el contenido del archivo SPI.h; y en el programa 12.3, el contenido del archivo SPI.c.

Programa 12.2. Contenido del archivo SPI.h

```
#define SPI_CLOCK_64 2
#define SPI_CLOCK_16 1
#define SPI_CLOCK_4  0
#define MODO_A          'A'
#define MODO_B          'B'
#define MODO_C          'C'
#define MODO_D          'D'
#define SDI_MIT         0
#define SDI_FIN         1
void INICIO_SPI(char reloj, char modo, char smp);
void TX_SPI(char dato);
char RX_SPI(char dato);
```

Programa 12.3. Contenido del archivo SPI.c

```
#include <xc.h>
#include "SPI.h"
void INICIO_SPI(char reloj, char modo, char smp)
{
	SSPSTATbits.SMP=smp;
	SSPCON1 = reloj;
	if(modo == 'A')
	{
		SSPSTATbits.CKE = 1;
		SSPCON1bits.CKP = 1;
	}
	else if(modo == 'B')
	{
		SSPSTATbits.CKE = 1;
		SSPCON1bits.CKP = 0;
	}
	else if(modo == 'C')
	{
		SSPSTATbits.CKE = 0;
		SSPCON1bits.CKP = 1;
	}
	else if(modo == 'D')
	{
		SSPSTATbits.CKE = 0;
		SSPCON1bits.CKP = 0;
	}
	SSPCON1bits.SSPEN = 1;
}
void TX_SPI(char dato)
{
	SSPBUF = dato;
	while(PIR1bits.SSPIF == 0);
	PIR1bits.SSPIF = 0;
}
char RX_SPI(char dato)
{
	SSPBUF = dato;
	while(PIR1bits.SSPIF == 0);
	PIR1bits.SSPIF = 0;
	dato = SSPBUF;
	return(dato);
}
```

La función INICIO_SPI() recibe tres parámetros: la frecuencia de reloj, el modo de generación de señal y la fase de muestreo de la línea SDI. La frecuencia de reloj puede tomar tres posibles valores: $\frac{F_{osc}}{64}$, $\frac{F_{osc}}{16}$ o $\frac{F_{osc}}{4}$. Para configurar estos parámetros se han definido las constantes SPI_CLOCK_64, SPI_CLOCK_16 y SPI_CLOCK_4, respectivamente. El segundo parámetro representa el modo, que puede tomar cuatro posibles valores, definidos también con los siguientes alias: MODO_A, MODO_B, MODO_C y MODO_D. Finalmente, el tercer parámetro puede tomar el valor SDI_MIT o SDI_FIN para indicar que la señal en el pin SDI se muestreará a la mitad del periodo de reloj o al final del periodo de reloj, respectivamente.

La función *TX_SPI()* permite transmitir por el bus SPI el parámetro ingresado a la función. Automáticamente, esta rutina genera los 8 pulsos de reloj en la línea SCK y produce la trama serial con el código binario del dato ingresado por la línea SDO.

Finalmente, la función *RX_SPI()* retorna el valor leído por la línea SDI que se encuentra en el registro SSPBUF. En un proceso de lectura, algunos dispositivos periféricos necesitan transmitir un byte sin uso por la línea SDO. Es decir, el dispositivo esclavo recibirá un código binario por la línea SDO que no le significará ninguna operación. En la mayoría de dispositivos, este código binario es el 0xFF, pero en otros, como el DS1305, el código binario es 0x00. La función *RX_SPI()* recibe como parámetro dicho byte a enviar durante el proceso de lectura. Luego, genera los 8 pulsos de reloj y lee el valor del buffer de recepción para luego retornarlo en la función.

12.5.1 Ejemplo de configuración de un reloj en tiempo real DS1305

El circuito integrado DS1305 de la empresa Maxim Dallas es un reloj en tiempo real capaz de actualizar la hora y la fecha con precisión de un segundo. El integrado posee un empaque tipo PDIP 16 y presenta algunas prestaciones que lo hacen bastante interesante.

El formato de hora y fecha llevado permite controlar fechas durante un periodo de 100 años (desde el año 2000 hasta el año 2099) y posee un mecanismo de corrección por año bisiesto. El DS1305 se puede configurar en un formato de hora con indicador AM/PM. Adicionalmente, incluye 96 bytes de memoria RAM para poder almacenar datos.

El integrado incluye tres pines para conexión de fuente externa: Vcc1, Vcc2 y Xbat. Vcc1 es el pin de alimentación primario y aquí se conecta un voltaje entre 2 y 5.5 V. Vcc2 es un pin especial para la conexión de una batería recargable.

Mientras la fuente en Vcc1 esté conectada, la batería recargable conectada a Vcc2 estará recibiendo carga de voltaje. El DS1305 posee un mecanismo de control de energía en el cual, en el momento de fallar o desactivarse la fuente en Vcc1, el voltaje de alimentación inmediatamente será extraído del pin Vcc2. Por otro lado, también se tiene el pin Xbat, que cumple la misma función que Vcc2, pero solamente permite la conexión de baterías no recargables (como es el caso de una batería de litio).

La fuente de voltaje se puede variar de 2 V a 5.5 V, con lo cual es posible interconectar el RTC con microcontroladores que trabajan con tensiones de 3.3 V, como lo es, por ejemplo, el MC9S12NE64 de Freescale. Además, el DS1305 incluye un pin Vccif que se puede conectar a un nivel de tensión de referencia para que las líneas de comunicación del RTC trabajen al mismo voltaje del microcontrolador.

El DS1305 posee dos alarmas configurables, las cuales trabajan directamente con dos pines INT0 e INT1 que generan un flanco de bajada (deben conectarse con una resistencia de *Pull Up* a uno

de los pines de interrupción externa del microcontrolador) cada vez que la condición configurada se ha generado. La alarma puede activarse cada segundo, cada minuto, cada hora o una vez al día[53].

El RTC maneja dos protocolos de comunicación para intercambiar datos con el microcontrolador: SPI y SPI Microwire. El primer protocolo es el que se ha mencionado en este capítulo e incluye cuatro pines (SDO, SDI, SCK y CS). El segundo trabaja con tres pines: SDO y SDI están unidos en un mismo nodo. Esta interfaz utiliza el mismo canal para transmisión y recepción de datos (Half Dúplex).

En el gráfico 12.7 se presenta la funcionalidad de cada pin del DS1305; y, en el gráfico 12.8, el diagrama de pines para el formato PDIP.

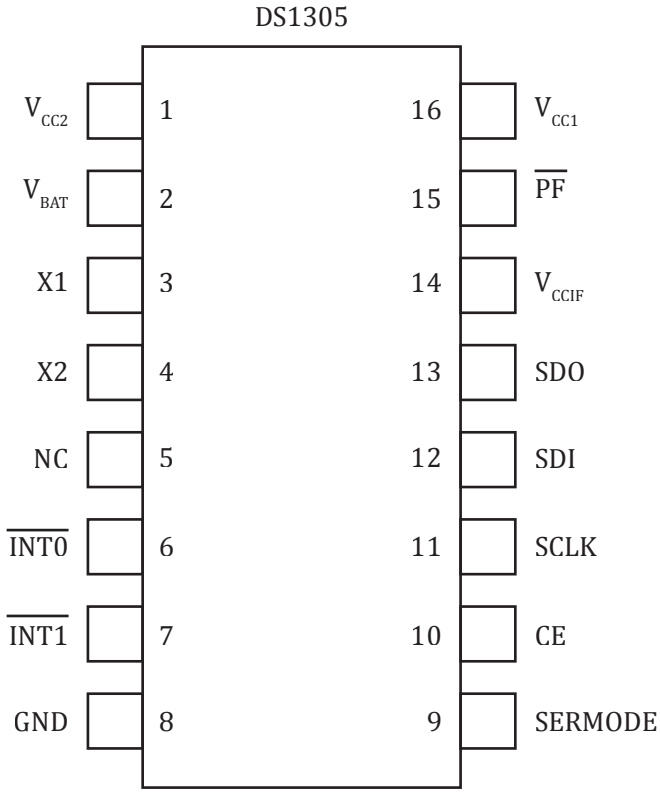
Gráfico 12.7. Funciones de los pines del circuito integrado DS1305

PIN	Nombre	Función
1	V_{CC2}	Fuente de alimentación secundaria. Es posible conectar a este pin una batería recargable que va a poder recargarse mediante el mecanismo de carga que posee el RTC.
2	V_{BAT}	Conexión para una batería de litio no recargable. Si este pin no se va a utilizar debe ser conectado a tierra.
3	X1	Conexión para una de las patas del cristal de 32.768 kHz.
4	X2	Conexión para la otra pata del cristal.
5	N.C	Sin conexión.
6	INT0	Indicador de interrupción por flanco de bajada. Esta interrupción se puede generar tanto por un evento en la Alarma 0 como en la Alarma 1. Debido a que posee una salida tipo drenador abierto requiere una resistencia de *Pull Up*.
7	INT1	Al igual que el pin INT0 puede generar interrupciones pero solo por eventos configurados en la Alarma 1.
8	GND	Tierra
9	SERMODE	Permite configurar el protocolo de comunicación utilizado. Si se conecta a Vcc, se selecciona el modo SPI. Si se conecta a tierra se seleccionará el modo SPI Microwire.
10	CE	Habilitador de transferencia. Este pin debe mantenerse en nivel '1' antes de cualquier transferencia.
11	SCLK	Señal de reloj del bus SPI. La frecuencia de esta señal no debe exceder los 2 MHz.
12	SDI	Entrada de datos del RTC. Se debe conectar al pin MOSI del microcontrolador.
13	SDO	Salida de datos del RTC. Se debe conectar al pin MISO del microcontrolador.
14	V_{CCIF}	Voltaje de referencia. Si el microcontrolador se encontrara operando a 3.3 V y la fuente de alimentación del RTC fuera de 5 V, este pin debería conectarse a 3.3 V para que las líneas de comunicación del puerto SPI sean compatibles con 3.3 V.
15	PF	Indicador de pérdida de la fuente de alimentación primaria. Al desconectarse la fuente V_{CC1} y empezar a trabajar con cualquiera de las dos fuentes alternativas, el DS1305 generará un flanco de bajada en este pin.
16	V_{CC1}	Fuente primaria de alimentación.

Elaboración propia.

53 Cfr. Cady 2006, revisado y modificado por Salas 2014.

Gráfico 12.8. Diagrama de pines del circuito integrado DS1305



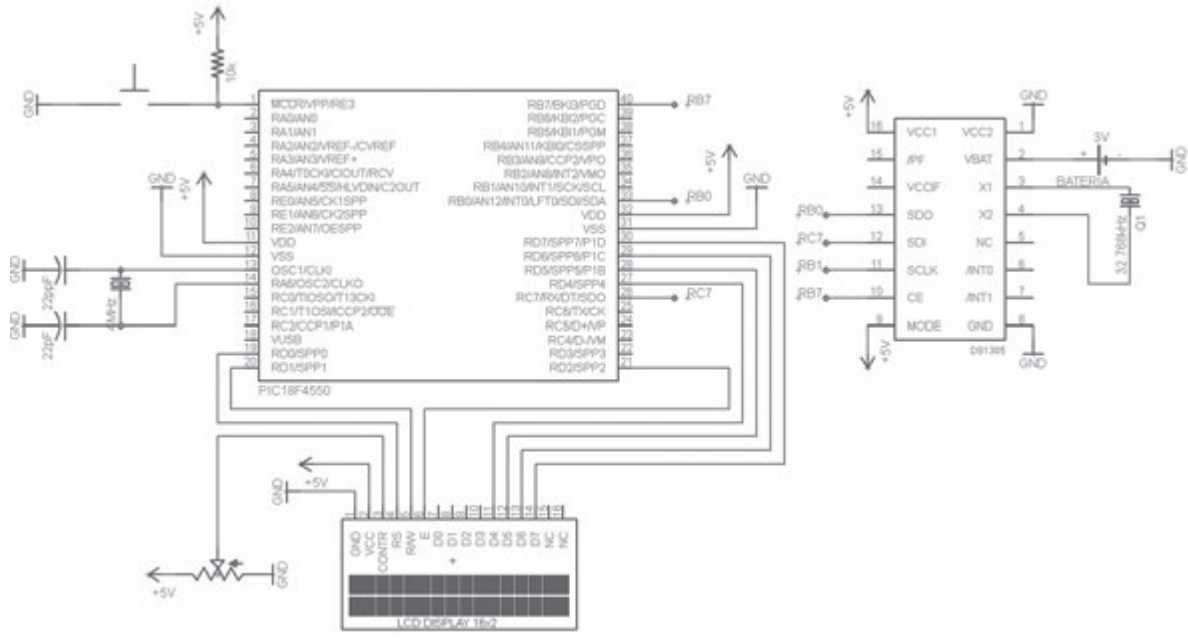
Elaboración propia.

La conexión más simple que se puede obtener entre el microcontrolador PIC18F4550 y el DS1305 se presenta en el gráfico 12.9. El pin SERMODE se coloca a 5 V para configurar el protocolo de comunicación como SPI. Las líneas de alimentación auxiliares se ponen a tierra, dado que no están siendo utilizadas.

Una vez alimentado, el primer paso consiste en configurar la fecha y la hora del DS1305 para que, posteriormente, las lecturas que se hagan del RTC entreguen la hora y la fecha actualizadas.

La actualización de la hora y la fecha sigue un formato que debe tomar en cuenta el contenido de los registros internos del RTC. El contenido de estos registros se muestra en el gráfico 12.10.

Gráfico 12.9. Diagrama de conexiones entre el PIC18F4550 y el DS1305



Elaboración propia.

Cada registro posee dos direcciones diferentes, una para escritura y otra para lectura. En otras palabras, para poder configurar el RTC es necesario transmitir primero la dirección y luego el dato. Por ejemplo, para poder configurar los segundos, primero se debe enviar la dirección (0x80) y luego el contenido del segundero.

Los registros se dividen en tres grupos diferentes. El primer grupo son los registros de actualización y lectura de fecha y hora (0x80-0x86, 0x00-0x06, respectivamente); el segundo corresponde a los registros para configuración de la Alarma 0 y la Alarma 1; y, finalmente, en el último grupo se encuentran los registros de control, estado, configuración de carga de batería y 96 bytes de RAM para almacenamiento de datos.

Para los registros de actualización de hora y fecha, el contenido de cada registro se guarda en formato BCD. Los cuatro bits menos significativos representan el formato BCD de la cifra menos significativa del día, el minuto, la hora, etc. De igual forma, los dos a tres bits más significativos representan el digito más significativo. El bit 7 siempre se encuentra en 0. Por ejemplo, para escribir el minuto 53 en el registro de minutos, primero se debe enviar la dirección de escritura 0x81 y luego el valor 0x53. Posteriormente, si se desea leer este valor más adelante, se deberá enviar el valor 0x01 y luego una operación de lectura por el bus SPI. Se debe tener presente que si la lectura se hace después de un minuto o más, el valor recibido ya no será 0x53, sino el valor anterior sumado con los minutos que se demoró en hacer la lectura desde la última escritura.

El bloque de registros de Alarma 0 y Alarma 1 permite configurar los pines INT0 e INT1 para que generen flancos de bajada cada vez que el contador de la hora es igual a la hora configurada en estos registros. El bit M permite desenmascarar el uso de cada registro, siendo '0' el valor que lo posibilita. Por otro lado, es muy importante la colocación de una resistencia de *Pull Up* en los pines INT0 e INT1, si es que estos van a ser utilizados.

Gráfico 12.10. Registros internos del circuito integrado DS1305

HEX ADDRESS READ	HEX ADDRESS WRITE	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	RANGE
00h	80h	0	10 Seconds			Seconds				00–59
01h	81h	0	10 Minutes			Minutes				00–59
02h	82h	0	12	P	10 Hour	Hours				01–12 + P/A
				A						00–23
			24	10						
03h	83h	0	0	0	0	Day				1–7
04h	84h	0	0	10 Date		Date				1–31
05h	85h	0	0	10 Month		Month				01–12
06h	86h	10 Year				Year				00–99
—	—	Alarm 0								—
07h	87h	M	10 Seconds Alarm			Seconds Alarm				00–59
08h	88h	M	10 Minutes Alarm			Minutes Alarm				00–59
09h	89h	M	12	P	10 Hour	Hour Alarm				01–12 + P/A
				A						00–23
			24	10						
0Ah	8Ah	M	0	0	0	Day Alarm				01–07
—	—	Alarm 1								—
0Bh	8Bh	M	10 Seconds Alarm			Seconds Alarm				00–59
0Ch	8Ch	M	10 Minutes Alarm			Minutes Alarm				00–59
0Dh	8Dh	M	12	P	10 Hour	Hour Alarm				01–12 + P/A
				A						00–23
			24	10						
0Eh	8Eh	M	0	0	0	Day Alarm				01–07
0Fh	8Fh	Control Register								—
10h	90h	Status Register								—
11h	91h	Trickle Charger Register								—
12h–1Fh	92h–9Fh	Reserved								—
20h–7Fh	A0h–FFh	96 Bytes User RAM								00–FF

Fuente: Hoja de datos del DS1305.

En cuanto a los registros de control, estado y memoria RAM, estos también pueden ser accedidos de forma directa. El registro de control es muy importante para la inicialización de toda la comunicación con el RTC. En el gráfico 12.11 se observa el contenido de este registro.

Gráfico 12.11. Registro de control del circuito integrado DS1305

$\overline{\text{EOSC}}$	WP	0	0	0	INTCN	AIE1	AIE0

Elaboración propia.

El bit EOSC permite activar el oscilador interno del RTC para que la cuenta del tiempo se pueda llevar a cabo. Para activar esta opción el bit EOSC debe ser '0'. Este bit no necesariamente va a ser '0' al encender la fuente del RTC, por lo que es fundamental poner a nivel '0' el bit antes de cualquier transferencia de información al DS1305.

El bit WP permite proteger al RTC contra escritura, evitando que cualquier transferencia de información del microcontrolador a través del puerto SPI sea procesada. Para eliminar la protección contra escritura, este bit WP deberá ser '0'.

El bit INTCN permite configurar los pines de interrupción que serán usados para indicar la activación de la Alarma 0 y la Alarma 1. Si este bit se encuentra en '0', tanto la Alarma 0 como la Alarma 1 generarán un flanco de bajada en el pin INT0. Si el bit INTCN es '1', entonces la Alarma 0 generará un evento en el pin INT0; y la Alarma 1, en el pin INT1.

Finalmente, los pines AIE1 y AIE0 permiten activar (con '1') la Alarma 1 y la Alarma 0, respectivamente.

La comunicación SPI se selecciona al conectar el pin SERMODE a Vcc. Esta comunicación utiliza 4 pines: SDI, SDO, SCK y CE, conectados al RTC según el esquema que se mostró en el gráfico 12.9.

El bit más significativo se transmite primero en la transferencia por el bus SPI. La frecuencia de reloj no debe exceder los 2 MHz y la polaridad del reloj se puede dejar en cualquier posición, ya que el DS1305 se puede adaptar a la polaridad de la señal de reloj del maestro.

La escritura de un byte de información se realiza enviando primero la dirección y luego el dato. El primer paso consiste en poner el pin CE a '1'; después se envían 8 pulsos de la señal de reloj sincronizados con el valor de la dirección en el pin SDO. Luego de terminados los 8 pulsos, se generan nuevamente 8 pulsos más enviando el dato que se quiere almacenar en el registro respectivo. Finalmente, el pin CE se pone a nivel '0'.

En el caso de una operación de lectura, el proceso es similar. Primero se habilita el RTC poniendo el pin CE a '1'. Luego, se envían 8 pulsos de reloj sincronizados con la dirección de lectura en el pin SDO. Después, se envían 8 pulsos más, pero se realiza la lectura del pin SDI y se recibe el dato en el registro SSPBUF. Finalmente, el pin CE se pone a nivel '0'.

En el programa 12.4 se muestra un ejemplo basado en el circuito esquemático del gráfico 12.9. En el siguiente programa se configura el RTC DS1305 para que parta de la hora 23:58:45 y con fecha inicial del martes 05 de junio del año 2014. Finalmente, se configurará una de las alarmas del RTC para que genere un pulso por la línea INT0 cada vez que todo minuto llegue a los 30 segundos (sin embargo, esta línea no se encuentra conectada. Podría conectarse al único pin de interrupción externa disponible RB2/INT2).

Una vez cargados los parámetros iniciales, el RTC comenzará a actualizar continuamente los valores cada segundo y de forma automática. El programa, entonces, se encargará de realizar una lectura del RTC cada 100 milisegundos, aproximadamente (aunque podría ser un tiempo mayor) para actualizar la hora y la fecha en la pantalla LCD, controlada a través del Puerto D haciendo uso de la librería LCD vista en los capítulos anteriores. En algunos casos (como en los registros de dirección 0x82 y 0x83) es necesario grabar un valor 0x00 antes de colocar el nuevo byte de configuración, para permitir que RTC configure la hora y el día de la semana.

Programa 12.4. Algoritmo de inicialización del DS1305 con una hora y fecha determinada y de actualización de la información en la pantalla LCD

```
#include <xc.h>
#include <plib/delays.h>
#pragma config FOSC    = XTPLL_XT
#pragma config PLLDIV = 1
#pragma config CPUDIV = OSC1_PLL2
#pragma config PWRT    = ON
#pragma config WDT      = OFF
#pragma config LVP       = OFF
#pragma config BOR       = OFF
#pragma config DEBUG   = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE   = ON

#define CE LATBbits.LATB7
#include "SPI.h"
#include "LCD.h"
typedef union{
        struct{
                unsigned char cifra0:4;
                unsigned char cifra1:4;
        }_byte;
        unsigned char data;
}dato_RTC;
dato_RTC dato;
unsigned char hor2=0,hor1=0,min2=0,min1=0,seg2=0,seg1=0;
unsigned char dia2=0, dia1=0, dia_sem2=0, dia_sem1=0, mes2=0, mes1=0, ano2=0, ano1=0;
void EscribeHora(void);
void main()
{
        TRISB = 0x01;
        TRISD = 0x08;
        TRISC = 0x7F;
        CE    = 0;            //Se desactiva el RTC
        Delay10KTCYx(240);
        LCD_CONFIG();
        INICIO_SPI(SPI_CLOCK_64,MODO_A,SDI_MIT);
        ESCRIBE_MENSAJE("Hora:",5);
        CURSOR_ONOFF(OFF);
```

```
EscribeHora();
CE    = 1;
TX_SPI(0x8F); //Dirección del registro de control
TX_SPI(0x01); //Deshabilitar WP, OSC ON, ALARMA INT0 ON
CE          = 0;
while(1)
{
        Delay10KTCYx(120);
        POS_CURSOR(1,6);
        ENVIA_CHAR(hor2+'0');
        ENVIA_CHAR(hor1+'0');
        ENVIA_CHAR(':');
        ENVIA_CHAR(min2+'0');
        ENVIA_CHAR(min1+'0');
        ENVIA_CHAR(':');
        ENVIA_CHAR(seg2+'0');
        ENVIA_CHAR(seg1+'0');
        POS_CURSOR(2,1);
        ENVIA_CHAR(dia_sem2+'0');
        ENVIA_CHAR(dia_sem1+'0');
        ENVIA_CHAR('-');
        ENVIA_CHAR(mes2+'0');
        ENVIA_CHAR(mes1+'0');
        ESCRIBE_MENSAJE("-20",3);
        ENVIA_CHAR(ano2+'0');
        ENVIA_CHAR(ano1+'0');
        CE = 1;
        TX_SPI(0x00);
        dato.data = RX_SPI(0x00);
        CE = 0;
        seg2 = dato._byte.cifra1;
        seg1 = dato._byte.cifra0;
        CE = 1;
        TX_SPI(0x01);
        dato.data = RX_SPI(0x00);
        CE = 0;
        min2 = dato._byte.cifra1;
        min1 = dato._byte.cifra0;
        CE = 1;
        TX_SPI(0x02);
        dato.data = RX_SPI(0x00);
```

```
                CE = 0;
                hor2 = dato._byte.cifra1;
                hor1 = dato._byte.cifra0;
                CE = 1;
                TX_SPI(0x03);
                dato.data = RX_SPI(0x00);
                CE = 0;
                dia_sem2 = dato._byte.cifra1;
                dia_sem1 = dato._byte.cifra0;
                CE = 1;
                TX_SPI(0x04);
                dato.data = RX_SPI(0x00);
                CE = 0;
                dia2 = dato._byte.cifra1;
                dia1 = dato._byte.cifra0;
                CE = 1;
                TX_SPI(0x05);
                dato.data = RX_SPI(0x00);
                CE = 0;
                mes2 = dato._byte.cifra1;
                mes1 = dato._byte.cifra0;
                CE = 1;
                TX_SPI(0x06);
                dato.data = RX_SPI(0x00);
                CE = 0;
                ano2 = dato._byte.cifra1;
                ano1 = dato._byte.cifra0;
        }
}
void EscribeHora(void)
{
        CE = 1;
        TX_SPI(0x8F);//Dirección del registro de control
        TX_SPI(0x01);//Deshabilitar WP, OSC ON, ALARMA INT0 ON
        CE = 0;
        Delay10TCYx(1);
        CE = 1;
        TX_SPI(0x80);//Dirección del registro de segundos.
        TX_SPI(0x45);//45 SEGUNDOS.
        CE=0;
        Delay10TCYx(1);
```

```
    CE=1;
    TX_SPI(0x81);  //Dirección del registro de minutos.
    TX_SPI(0x58);  //58 MINUTOS.
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x82);  //Dirección del registro de horas.
    TX_SPI(0x00);  //SELECCION DEL MODO 0-23 HORAS.
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x82);  //Dirección del registro de horas.
    TX_SPI(0x23);  //CONFIGURACION DE LAS 23 HORAS.
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x83);  //Dirección del registro día de la semana.
    TX_SPI(0x00);  //Limpieza del registro
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x83);  //Dirección del registro día de la semana.
    TX_SPI(0x03);  //03 DIA MARTES
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x84);  //Dirección del registro día del mes.
    TX_SPI(0x05);  //05 DIA DEL MES
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x85);  //Dirección del registro mes del año.
    TX_SPI(0x06);  //06 JUNIO
    CE=0;
    Delay10TCYx(1);
    CE=1;
    TX_SPI(0x86);  //Dirección del registro del año.
    TX_SPI(0x14);  //14 AÑO 2014
    CE=0;
    Delay10TCYx(1);
//Alarma 0 cada 60 segundos.
```

```
        CE=1;
        TX_SPI(0x87);  //Dirección de la alarma para los segundos.
        TX_SPI(0x30);  //cada vez que la hora marque 30 segundos se generará una
                       //INT0.
        CE=0;
        Delay10TCYx(1);
        CE=1;
        TX_SPI(0x88);  //Dirección de la alarma para los minutos.
        TX_SPI(0x80);  //Máscara de interrupción activada. OFF.
        CE=0;
        Delay10TCYx(1);
        CE=1;
        TX_SPI(0x89);  //Dirección de la alarma para las horas.
        TX_SPI(0x80);  //Máscara de interrupción activada. OFF.
        CE=0;
        Delay10TCYx(1);
        CE=1;
        TX_SPI(0x8A);  //Dirección de la alarma para el día.
        TX_SPI(0x80);  //Máscara de interrupción activada. OFF.
        CE=0;
}
```

12.5.2 Ejemplo de control de una matriz de LED usando el MAX6952

Muchas empresas tienen la necesidad de mostrar información importante que se encuentre a la vista de las personas, tanto del personal laboral como de la clientela. La información no requiere ser mostrada por completo de una sola vez y puede estar desplazándose carácter por carácter. La temperatura, la fecha, la humedad y otros datos pueden ser mostrados. Por ejemplo, los colegios requieren mostrar los rivales que van a enfrentar en competencias, los resultados de juegos anteriores u otros eventos importantes, utilizando paneles grandes. Muchos de estos paneles utilizan matrices de LED, debido al brillo y versatilidad.

Las matrices de LED se caracterizan por su número de filas y columnas. El tipo más popular de matriz posee cinco columnas y siete filas (5 x 7). Otras configuraciones, como de 5x8 y 8x8, también se encuentran disponibles. Es obvio que mientras más filas y columnas presente la matriz, mejor resolución tendrá.

Al igual que los Displays 7 segmentos, existen matrices de colores verde, amarillo, rojo, bicolor (rojo y verde) y tricolor (rojo, verde y azul).

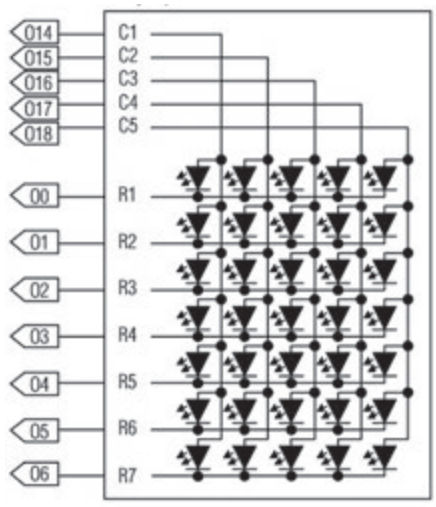
Las matrices de LED se organizan en filas de cátodos (columna de ánodos) o fila de ánodos (columna de cátodos). En la organización de fila de cátodos, los LED en una fila tienen el cátodo común. En una organización de fila de ánodos, los LED en una fila presentan el ánodo común. La organización de fila de cátodos se muestra en el gráfico 12.12.

Las dimensiones de una matriz de LED frecuentemente se encuentran indicadas por su altura. Las alturas más comunes para una matriz de LED incluyen 0.7'', 1.2'', 1.7'' y 2.3''. De igual manera, existen tipos de matrices de mayores dimensiones en el mercado.

Debido a la conexión interna de una matriz de LED, son necesarios dos puertos paralelos para poder realizar el control. La matriz de LED solo puede encender una fila a la vez, de manera que, conmutando rápidamente el encendido desde la fila de arriba hacia la fila de abajo (o de abajo para arriba), se logra visualizar un carácter estático. Normalmente, múltiples matrices de LED son necesarias en las aplicaciones; así, se requiere una técnica de multiplexaje para el encendido de cada fila o columna de LED, y esto exige tiempo del CPU.

Debido a la popularidad de las matrices de LED, algunas empresas han producido circuitos integrados especiales para poder controlarlas. Una de ellas es Maxim Dallas, que provee circuitos integrados con protocolo SPI (MAX6952) o I^2C (MAX6953) para matrices de fila de cátodos de dimensión 5x7.

Gráfico 12.12. Matriz de LED de 8 filas de cátodos y 5 columnas de ánodos



Fuente: Hoja de datos del MAX6952.

El MAX6952 está diseñado para controlar una matriz de LED de fila de cátodos con una organización de 5x7. Este chip puede operar con una fuente de voltaje desde 2.7 a 5.5 V, y puede controlar cuatro matrices monocolor o dos matrices bicolores al mismo tiempo. El chip posee internamente una memoria ROM con un formato de caracteres ASCII de 104 elementos y un espacio de memoria RAM para la creación de hasta 24 caracteres definidos por el usuario. Los caracteres en ROM siguen el formato de letra tipo Arial, con la suma de algunos símbolos comunes como €, £, μ, ¥, ±, ↑ y ↓. Los 24 caracteres definibles por el usuario son guardados por el usuario en la memoria RAM interna del MAX6952 a través de la interface serial, y se pierden cada vez que se le quita la energía al integrado. Este chip permite la oscilación automática para cada segmento y provee un control digital de 15 niveles de brillo. Adicionalmente, el integrado presenta el modo *Test*, en el cual todos los LED son forzados a encender[54].

Las funciones de los pines son descritas en el gráfico 12.13. Cabe mencionar que el tipo de empaque que se está referenciando es el PDIP 40.

54 Cfr. Cady 2006, revisado y modificado por Salas 2014.

Gráfico 12.13. Funciones de los pines del MAX6952

Nombre	PIN	Función
O0 a O13	1,2,3,7-15,26,27	Drivers de cátodos de LED. O0 a O13 llevan la corriente desde las filas de cátodos.
GND	4,5,6,18	Tierra
ISET	17	Configuración de la corriente de segmento. Se debe conectar ISET a tierra a través de una resistencia R_{SET} para configurar el pico de corriente.
BLINK	19	Salida de reloj de oscilación. La salida es Drenador abierto.
DIN	20	Entrada serial de datos. Los datos son cargados en el registro interno de 16 bits en el flanco de subida del reloj.
CLK	21	Reloj serial de entrada. En el flanco de subida los datos se cargan el en registro interno de recepción. Durante los flancos de bajada del CLK los datos son enviados por DOUT.
DOUT	22	Salida serial de datos. Los datos ingresados en DIN son enviados por el pin 15 cinco ciclos de reloj después. Los bits de datos se envían en los flancos de bajada de CLK.
CS	23	Selector de Chip. Cuando este pin se pone a 0 el MAX6952 se habilita.
OSC	24	Entrada de reloj multiplexada. Para utilizar el oscilador interno, se debe conectar el condensador CSET de OSC a tierra. Para utilizar el reloj interno se debe conectar el pin OSC a un reloj CMOS de 1 MHz a 8 MHz.
O14 a O23	28-34,38,39,40	Drivers de ánodo de los LED. O14 a O23 son fuentes de corriente para las columnas de ánodos de la matriz.
V+	35,36,37	Fuente positiva de voltaje. Se debe conectar un condensador de paso de 47 uF electrolítico y un condensador de 0.1 uF.

Elaboración propia.

La típica conexión que maneja cuatro dígitos de matrices monocolores se presenta en el gráfico 12.14. En el gráfico 12.15 se observa el diagrama de conexión de las cuatro matrices de LED monocolor con el MA6952.

Gráfico 12.14. Esquema de conexión de cuatro matrices monocolor al MAX6952

Dígito	00-06	07-013	014-018	019-023
1	Filas del dígito 0 (cátodos) R1 a R7. Filas del dígito 1 (cátodos) R1 a R7.	-	Columnas del dígito 0 (ánodos) C1 a C5.	Columnas del dígito 1 (ánodos) C6 a C10.
2	-	Filas del dígito 2 (cátodos) R1 a R7. Filas del dígito 3 (cátodos) R1 a R7.	Columnas del dígito 2 (ánodos) C1 a C5.	Columnas del dígito 3 (ánodos) C6 a C10.

Elaboración propia.

El MAX6952 se conecta al microcontrolador a través de cuatro pines: CLK, CS, DIN y DOUT. La señal CS debe estar en bajo para poder enviar o recibir datos desde el sistema embebido. La señal en DIN debe estar sincronizada con el flanco de subida del reloj. El MAX6952 puede enviar los datos hasta una frecuencia de 26 MHz. Cuando el MAX6952 no está siendo accedido, el pin DOUT no se encuentra en alta impedancia, al contrario de las especificaciones del protocolo SPI, por lo cual este integrado no debería compartir el bus con dispositivos SPI de otro tipo.

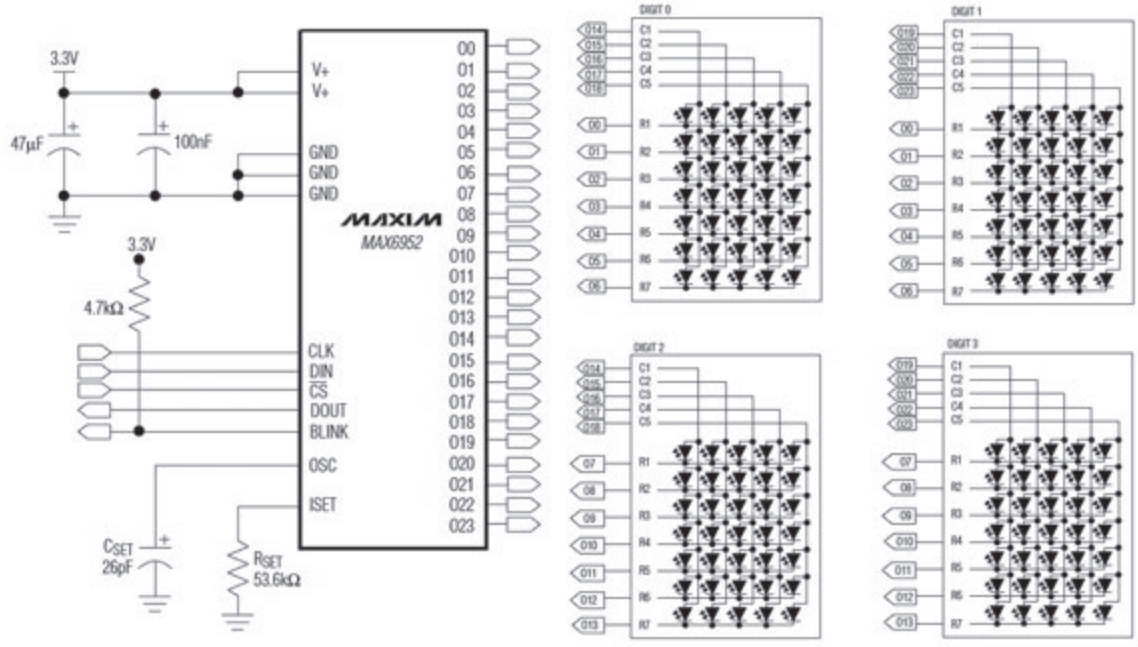
Múltiples MAX6952 pueden ser conectados en cascada a través de los pines DOUT de un dispositivo y DIN del siguiente chip. Las líneas CLK y CS se conectan en paralelo, como se observa en el gráfico 12.16. Los datos en DIN se propagan a través del registro de desplazamiento interno y aparece en DOUT 5 ciclos de reloj después, sincronizado con los flancos de bajada de CLK.

Cuando se envían comandos a la conexión en cascada de MAX6952, todos los dispositivos se acceden al mismo tiempo. Un acceso requiere 16 x n ciclos de reloj, donde n es el número de chips MAX6952 conectados en cascada. Para configurar solo un dispositivo conectado en la cadena, el usuario puede enviar el comando de no operación (0x00) a los otros.

El diagrama de bloques del MAX6952 se muestra en el gráfico 12.17. El MAX6952 contiene un registro de desplazamiento de 16 bits dentro del cual los datos han ingresado por el pin DIN siendo sincronizados con el flanco de subida de la señal de reloj, mientras la señal CS se encuentra en estado bajo. Los 16 bits del registro de desplazamiento se guardan en un registro *Latch* en el flanco de subida de la señal CS. Los 16 bits en el *Latch* son decodificados y ejecutados. La entrada de reloj se debe configurar en estado *Idle* bajo. Esta señal siempre debe estar en bajo antes de que se inicie una transferencia[55].

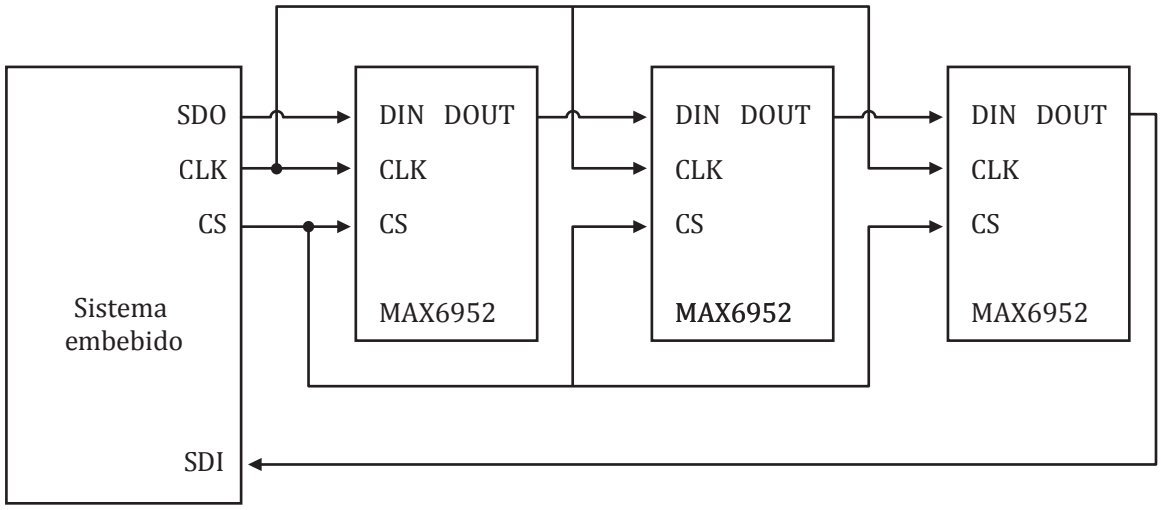
55 Cfr. Cady 2006, revisado y modificado por Salas 2014.

Gráfico 12.15. Diagrama de conexiones del MAX6952



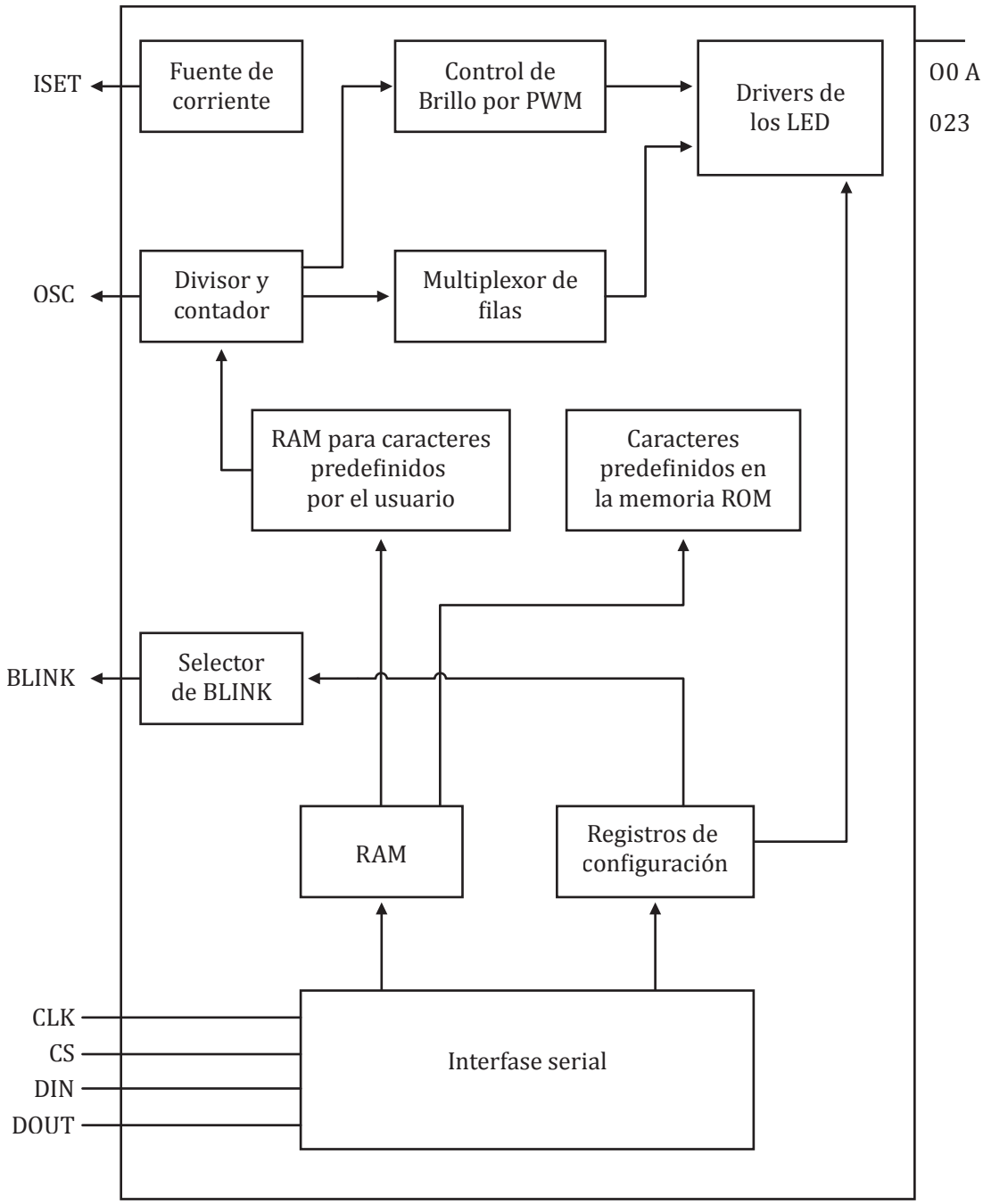
Fuente: Hoja de datos del MAX6952.

Gráfico 12.16. Diagrama de una conexión en cascada entre el sistema embebido y más de un MAX6952



Elaboración propia.

Gráfico 12.17. Diagrama de bloques de los elementos que conforman el MAX6952



Elaboración propia.

Los 8 bits más significativos del registro de desplazamiento seleccionan el registro destino. Dentro de este registro se ingresará el valor de los 8 bits menos significativos del registro de desplazamiento. El mapa de direcciones del MAX6952 se muestra en el gráfico 12.18.

Gráfico 12.18. Mapa de direcciones de los registros internos del MAX6952

Registro	Dirección (byte de comando) D15	D14	D13	D12	D11	D10	D9	D8	Código hexadecimal
No operación	R/W	0	0	0	0	0	0	0	0x00
Intensidad 10	R/W	0	0	0	0	0	0	0	0x01
Intensidad 32	R/W	0	0	0	0	0	0	0	0x02
Límite	R/W	0	0	0	0	0	0	0	0x03
Configuración	R/W	0	0	0	0	0	0	0	0x04
Caracteres definidos por el usuario	R/W	0	0	0	0	0	0	0	0x05
Reservado	R/W	0	0	0	0	0	0	0	0x06
Test de Matriz	R/W	0	0	0	0	0	0	0	0x07
Dígito 0 Plano P0	R/W	0	0	0	0	0	0	0	0x20
Dígito 1 Plano P0	R/W	0	0	0	0	0	0	0	0x21
Dígito 2 Plano P0	R/W	0	0	0	0	0	0	0	0x22
Dígito 3 Plano P0	R/W	0	0	0	0	0	0	0	0x23
Dígito 0 Plano P1	R/W	0	0	0	0	0	0	0	0x40
Dígito 1 Plano P1	R/W	0	0	0	0	0	0	0	0x41
Dígito 2 Plano P1	R/W	0	0	0	0	0	0	0	0x42
Dígito 3 Plano P1	R/W	0	0	0	0	0	0	0	0x43
Escribir el Dígito 0 en Plano P0 y P1 con los mismos datos	R/W	0	0	0	0	0	0	0	0x60
Escribir el Dígito 1 en Plano P0 y P1 con los mismos datos	R/W	0	0	0	0	0	0	0	0x61

Registro	Dirección (byte de comando) D15 D14 D13 D12 D11 D10 D9 D8	Código hexadecimal
Escribir el Dígito 2 en Plano P0 y P1 con los mismos datos	R/W 0 0 0 0 0 0 0	0x62
Escribir el Dígito 3 en Plano P0 y P1 con los mismos datos	R/W 0 0 0 0 0 0 0	0x63

Elaboración propia.

El procedimiento para escribir en el MAX6952 es el siguiente[56]:

1. El pin CLK del puerto SPI debe estar en bajo por defecto.
2. El pin CS se pone a cero para habilitar el registro de desplazamiento interno de 16 bits del MAX6952.
3. Enviar los datos desde el microcontrolador al pin DIN del MAX6952. El bit más significativo de la muestra se debe enviar primero y debe ser '0' para una operación de escritura.
4. El pin CS se pone en nivel alto.
5. El pin CLK se pone en bajo nuevamente.

Cualquier registro interno del MAX6952 puede ser leído al poner el bit D15 en '1'. El proceso de lectura es el siguiente:

1. El pin CLK debe estar en nivel bajo.
2. El pin CS debe estar en nivel bajo.
3. Enviar 16 bits por el pin DIN empezando por el bit 15. El bit 15 debe ser '1'. Entre los bits 14 y 8 se encuentra la dirección del registro a ser leído. El contenido de los bits 7 a 0 no interesa.
4. La señal CS se debe poner a '1'. Esto hará que los bits 7 a 0 del registro de desplazamiento se carguen con el contenido del registro especificado por la dirección.
5. CLK se pone a nivel bajo.
6. Se debe llevar a cabo otro comando de lectura (el cual puede ser el de no operación) y se examina el contenido de bits transferido por el pin DOUT. El segundo bloque de 8 bits de la trama recibida corresponde al contenido del registro enviado en la dirección anterior en el paso 3.

El MAX6952 utiliza 8 registros de dígitos para guardar los caracteres que el usuario desea mostrar en las cuatro matrices de 5x7 LED. Estos registros de dígitos son implementados con dos planos de 4 bytes, denominados P0 y P1. Cada dígito es representado por 2 bytes de memoria, un byte en el plano P0 y el otro en el plano P1. Los registros de dígito son mapeados de tal manera que un dato de

56 Cfr. Cady 2006.

dígito puede ser actualizado en el plano P0, P1 o en ambos al mismo tiempo, como se muestra en el gráfico 12.19.

Gráfico 12.19. Configuración del registro de dígito

Bit de segmento configurado en el Plano P1	Bit de segmento configurado en el Plano P0	Comportamiento del segmento
0	0	Segmento apagado.
0	1	Segmento encendido solo durante la primera mitad de cada periodo de *blinking.*
1	0	Segmento encendido solo durante la segunda mitad de cada periodo de *blinking.*
1	1	Segmento encendido.

Elaboración propia.

Los datos en los registros de dígito no controlan los segmentos de dígito directamente. En cambio, los registros de dato se usan para direccionar el generador de caracteres, el cual almacena los datos en un formato ASCII de 128 caracteres. Los 7 bits menos significativos del dígito de datos (D6 a D0) seleccionan el carácter. El bit más significativo del registro de datos (D7) selecciona cuando el carácter será usado directamente (D7='0') o cuando el carácter será invertido (D7='1'). La característica de inversión puede ser usada para mejorar la apariencia de las matrices bicolores, por ejemplo, mostrando un carácter rojo en un fondo verde.

El registro de configuración es usado para ingresar y salir del apagado, seleccionar la velocidad de *Blinking*, habilitar o deshabilitar la función de *Blink*, borrar dígitos de datos e inicializar el tiempo de *Blink*. El contenido del registro de configuración se observa en el gráfico 12.20.

Gráfico 12.20. Registro de configuración del MAX6952

P	X	R	T	E	B	X	S

Elaboración propia.

El bit P es el selector de lectura de la fase de *Blink*. El nivel '0' representa la fase de *Blink* P1 y el nivel '1' la fase de Blink P0.

El bit R permite controlar el borrado del dígito mostrado en la matriz de LED. El nivel '0' implica que no existe ningún efecto en los planos P0 y P1, mientras que el nivel '1' implica que los dígitos de ambos planos se eliminan.

El bit T configura la sincronización del tiempo de *Blink* global. Un nivel '0' en este bit supone que el contador de temporización del *Blink* global no queda afectado, mientras que un nivel '1' implica que los contadores de temporización quedan inicializados en el flanco de subida de la señal del pin CS.

El bit E habilita o deshabilita el *Blink*. Con un nivel '0', el *Blink* queda deshabilitado; y con un nivel '1', este se habilita.

El bit B permite seleccionar la velocidad de *Blink*. Si este bit es '0', se selecciona la oscilación lenta con un refresco de 1 segundo por plano. Si el bit es '1', se selecciona la opción de *Blink* rápido.

Finalmente, el bit S configura el modo de apagado. Con un nivel '0', el modo permanece apagado; y con un nivel '1', el modo es normal.

El brillo de la matriz de LED es definido por cuatro moduladores de ancho de pulso, uno para cada dígito de la matriz. Cada dígito es controlado por un *nibble* de uno de los dos registros de intensidad, Intensidad10 e Intensidad32. El *nibble* más significativo del registro Intensidad10 controla la intensidad de la matriz de LED 1, mientras que el *nibble* menos significativo del mismo registro controla la intesidad de la matriz de LED 0. Las matrices de LED 3 y 2 son controlados por los *nibbles* más significativos y menos significativos del registro *Intensidad32*, respectivamente. El modulador escala la corriente promedio por segmento en 16 pasos desde un máximo de 15/16 hasta un mínimo de 1/16 del pico de corriente. El tiempo mínimo de «blanqueo» entre dígitos es de 1/16 de un ciclo. El máximo Duty cycle es de 15/16.

El registro de límite configura cuantos dígitos monocolor se van a mostrar, tanto para las dos o cuatro matrices. Un dígito bicolor es conectado como dos dígitos monocromático. El esquema de multiplexaje controla los dígitos 0 y 1 al mismo tiempo, luego, los dígitos 2 y 3 a la vez. Para incrementar la efectividad del brillo en la matriz, el MAX6952 solamente maneja dos dígitos en vez de cuatro. Haciendo esto, la corriente promedio se duplica.

El contenido del registro límite se muestra en el gráfico 12.21. Este registro tiene solamente un bit implementado. Cuando este bit es '0', solo los dígitos 0 y 1 son mostrados. En cambio, cuando el bit es '1', se muestran los cuatro dígitos.

Gráfico 12.21. El registro límite del MAX6952

X	X	X	X	X	X	X	2 o 4

Elaboración propia.

El bit 2 o 4 permite mostrar solo dos dígitos en las matrices o los cuatro dígitos completos. Si este bit es '0', solo se mostrarán los dígitos 0 y 1. Si el bit es '1', se mostrarán los dígitos 0, 1, 2, y 3.

El registro de *Test* cambia a los drivers de LED en dos modos diferentes: Normal y *display test*. En el modo *display test* todos los LED se encienden. El contenido del registro *Test* se muestra en el gráfico 12.22.

Gráfico 12.22. El registro Test del MAX6952

X	X	X	X	X	X	X	Test

Elaboración propia.

Si el bit *TEST* permanece en nivel '0', la operación del MAX6952 será normal, mostrando los caracteres que el sistema embebido genere. En cambio, si este bit es '1', se ingresará a un estado *Test* en el cual aparecerá una secuencia predefinida por el fabricante en las matrices de LED.

La matriz de caracteres es de 5 x 7. El generador de caracteres comprende 104 caracteres en ROM y 24 caracteres definibles por el usuario. La selección de un total de 128 caracteres es representada por los siete bits menos significativos de los registros de 8 bits. El mapa de caracteres sigue el formato Arial para los 96 caracteres en el rango de 0x28 y 0x7F. Los primeros 32 caracteres del mapa contienen a las 24 posiciones definibles por el usuario (RAM00 a RAM23), más 8 caracteres comunes extra en ROM. Cuando el bit más significativo es '0', el dispositivo mostrará el formato de letra normal. En caso contrario, el MAX6952 mostrará el formato inverso.

Los 24 caracteres definidos por el usuario son representados por 120 entradas de 7 bits de datos. Son cinco entradas por carácter y estas se guardan en la RAM interna del MAX6952. Los 120 datos se escriben y leen a través de un solo registro en la dirección 0x05. El acceso al grabado de los datos se hace a través de un registro puntero de direcciones que posee la propiedad de autoincremento. El puntero de direcciones puede ser escrito configurando cada uno de las direcciones entre 0x00 y 0xF7, pero no puede ser leído.

Para definir nuevos caracteres, el usuario necesita configurar el puntero de direcciones. Esto se lleva a cabo al poner la dirección en el registro puntero de direcciones y colocando el bit 7 a '1'. Después de esto, es posible escribir el carácter en los 7 bits más bajos.

El puntero de direcciones se autoincrementa después de un acceso válido a los caracteres definibles. El autoincremento permite la entrada de 120 datos de forma muy rápida, debido a que el puntero de direcciones necesita configurarse solamente una vez. Cuando la última ubicación de datos 0xF7 es escrita, el puntero de direcciones incrementa hasta el valor 0x80 automáticamente.

El mapa de memoria de los caracteres definidos por el usuario se detalla en el gráfico 12.23. El comportamiento del puntero de direcciones se presenta en el gráfico 12.24. Para mostrar un carácter definido por el usuario, se debe enviar la dirección de la RAM desde 0x00 hasta 0x17, correspondiente a la dirección del puntero de direcciones, que es 5 por la dirección de la RAM (ya que un carácter utiliza 5 bytes)[57].

Gráfico 12.23. Mapa de memoria de caracteres definidos por el usuario en el registro 0x05

Código de dirección	Registro de datos	Lectura o escritura SPI	Función
0x85	0x00-0x7F	Lectura	Lectura de los 7 bits definibles por el usuario de la dirección respectiva. El bit MSB siempre es cero. El registro puntero de direcciones es incrementado después de una lectura.
0x05	0x00-0x7F	Escritura	Escritura de los 7 bits definibles por el usuario. El puntero de direcciones se incrementa después de la escritura.
0x05	0x80-0xFF	Escritura	Escribir en el puntero de direcciones con el registro de datos.

Elaboración propia.

57 Cfr. Cady 2006.

Gráfico 12.24. Comportamiento del puntero de direcciones

Puntero de direcciones	Acción
0x80-0xF6	Rango válido para configurar el puntero de direcciones. El puntero se autoincrementa después de una operación de lectura o escritura.
0xF7	Inicialización del puntero de direcciones al valor 0x80.
0xF8 a 0xFF	Rango no válido para configurar el puntero de direcciones.

Elaboración propia.

Cuando se activa el *Blinking* se logra que los LED cambien el contenido a mostrar del plano P0 a P1. Si el dato en el registro de dígito es diferente en dos planos, entonces este dígito aparecerá como si cambiara de un carácter a otro. Para hacer que un carácter aparezca y desaparezca de la matriz, se debe escribir el código del carácter en un plano y en el otro plano se debe escribir un carácter en blanco (0x20). Una vez que el *Blinking* ha sido configurado, este seguirá trabajando sin ninguna intervención adicional del microcontrolador. El *Blinking* es configurado al poner a '1' el bit E del registro de configuración.

La velocidad de *Blinking* puede ser programada para ser rápida o lenta, y es determinada por la frecuencia del reloj OSC y la puesta a '1' del bit de configuración B.

Cuando múltiples dígitos son mostrados, es posible seleccionar el sincronismo del *Blinking* de cada dígito. Internamente, la sincronización del *Blinking* es llevada a cabo inicializando la secuencia de multiplexaje. Así como los MAX6952 se pueden conectar en cascada con cada pin DOUT conectado al pin DIN del siguiente chip, la sincronización global también se puede llevar a cabo mediante el cambio del pin CS para cada dispositivo, tanto todos juntos como en una sucesión rápida.

El oscilador RC utiliza un resistor externo, R_{SET} y un capacitor externo C_{SET} para configurar la frecuencia del oscilador f_{OSC}. El rango permitido para f_{OSC} es desde 1 MHz hasta 8 MHz. R_{SET} también configura el pico de corriente del segmento. Los valores recomendados para R_{SET} y C_{SET} configuran el oscilador a 4 MHz, lo cual mantiene las frecuencias lenta y rápida a 0.5 Hz y 1 Hz, respectivamente. El valor recomendado para que R_{SET} configure el pico de corriente a 40 mA, lo cual hace que la corriente de segmento (I_{SEG}) sea ajustable desde 2.5 mA hasta 37.5 mA en pasos de 2.5 mA, es el siguiente:

$$I_{SEG} = \frac{K_I}{R_{SET}} mA$$

$$f_{OSC} = \frac{K_F}{(R_{SET} \times C_{SET} + C_{STRAY})} MHz$$

donde

$K_I = 2144$
$K_F = 6000$
R_{SET} = resistencia externa en kΩ
C_{SET} = condensador externo en pF
C_{SET} = capacitancia parásita entre el pin OSC y tierra. Normalmente es 2 pF.

El valor recomendado para R_{SET} es de 53.6 kΩ, y el valor recomendado para C_{SET} es de 26 pF.

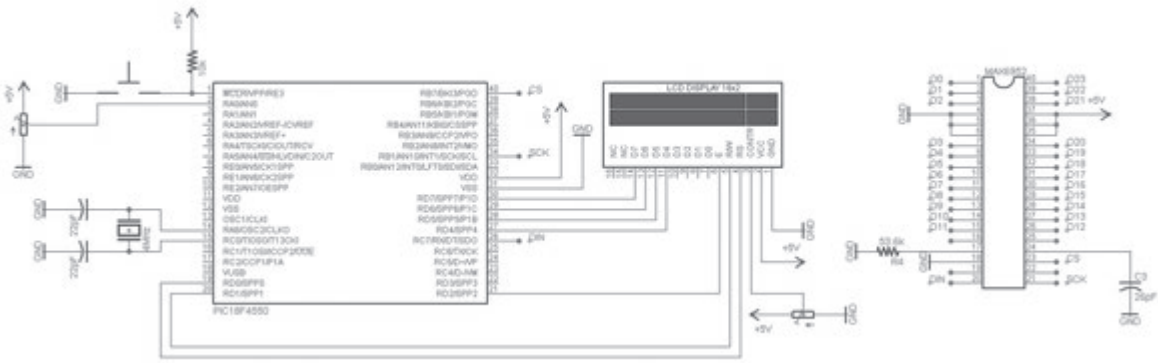
En el gráfico 12.25 se muestra la conexión entre el PIC18F4550 y el MAX6952 propuesta. El MAX6952 se encuentra conectado a cuatro matrices de LED. En el programa 12.5 se ofrece un ejemplo de control de los MAX6952 haciendo uso del PIC18F4550 y se genera un mensaje realizando un corrimiento de derecha a izquierda en las cuatro matrices de LED y en las cuatro primeras columnas de la pantalla LCD. El desplazamiento de cada carácter se realiza en un periodo de 200 milisegundos.

Programa 12.5. Algoritmo para generar un mensaje con desplazamiento a la izquierda en una matriz de LED mediante el MAX6952

```
#include <xc.h>
#include <plib/delays.h>
#pragma config PLLDIV = 1
#pragma config CPUDIV = OSC1_PLL2
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    = OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON
#define CS LATBbits.LATB7
#include "SPI.h"
#include "LCD.h"
void enviarmax(char direccion,char dato);
char mensaje[]="Programar el modulo SPI del PIC18F4550 es sencillo....";
unsigned int longitud;
unsigned char i=0;
void main()
{
        TRISCbits.TRISC7 = 0;
        TRISB = 1;
        TRISD = 0;
        Delay10KTCYx(240);
        LCD_CONFIG();
        INICIO_SPI(SPI_CLOCK_16,MODO_B,SDI_MIT);
        enviarmax(0x01,0xFF); //Intensidad para los dígitos 0 y 1.
        enviarmax(0x02,0xFF); //Intensidad para los dígitos 2 y 3.
        enviarmax(0x03,0x01); //Mostrar cuatro dígitos.
        enviarmax(0x04,0x11); //Registro de configuración
        enviarmax(0x07,0x00); //Deshabilitar el test
        longitud=strlen(mensaje);
```

```
        while(1)
        {
                enviarmax(0x20,mensaje[i]);
                POS_CURSOR(1,0);
                ENVIA_CHAR(mensaje[i]);
                enviarmax(0x21,mensaje[i+1]);
                ENVIA_CHAR(mensaje[i+1]);
                enviarmax(0x22,mensaje[i+2]);
                ENVIA_CHAR(mensaje[i+2]);
                enviarmax(0x23,mensaje[i+3]); //Valor dígito 3
               ENVIA_CHAR(mensaje[i+3]);
                i++;
              if(i>=longitud-3) i=0;
              Delay10KTCYx(240);
        }
}
void enviarmax(char direccion, char dato)
{
 CS = 0;
 TX_SPI(direccion);
 TX_SPI(dato);
 CS = 1;
}
```

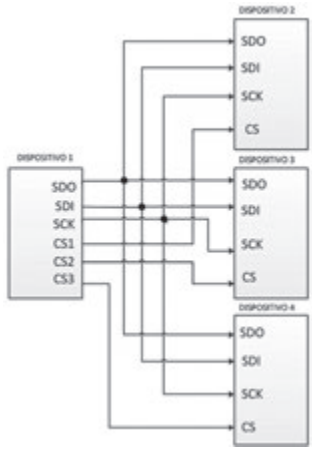
Gráfico 12.25. Circuito esquemático de la conexión entre el PIC18F4550 y el MAX6952 y la pantalla LCD



Elaboración propia.

Preguntas de repaso

1. Con respecto al protocolo SPI, responda lo siguiente:
   a. ¿Es un protocolo Full dúplex o Half dúplex? Justifique.
   b. ¿De qué parámetro depende la velocidad de transmisión?
2. Explique cuál es la función del pin CS en el protocolo de comunicación SPI.
3. Si una transferencia de datos por el protocolo SPI se realiza a una tasa de 100 kbps, ¿cuál es el periodo de la señal de reloj usada en la línea SCK?
4. Explique. ¿Cuáles son los modos de configuración de los estados de *Idle* y fase de generación de la señal SDO en el protocolo SPI?
5. En la siguiente figura se tienen cuatro dispositivos interconectados mediante el protocolo SPI.
   a. Indique cuál de los cuatro dispositivos cumple el rol de maestro y por qué.
   b. Si la señal de reloj generada por el pin SCK tiene una duración total de 80 µsegundos, ¿cuál es la tasa de bit a la que transmite el bus SPI?



Respuestas

Respuestas del capítulo 12:

http://bit.ly/1QmjqYQ

Capítulo 13. La interfaz serial I²C

Hasta el momento se han visto interfaces de comunicación asíncronas y síncronas en formato Full dúplex (como es el caso del protocolo SPI). A continuación se presenta el formato de comunicación serial síncrono I²C que corresponde al tipo Half dúplex. Es decir, la información en este protocolo viaja en dos sentidos diferentes en la línea de datos y requiere de un mecanismo de sincronismo para que el transmisor y el receptor se pongan de acuerdo en el momento del envío y la recepción.

La gran ventaja del bus I²C sobre el resto de protocolos es que solo requiere de 2 líneas de comunicación para enlazarse con diversos dispositivos conectados en el mismo bus. El I²C es un protocolo para cortas distancias dentro del mismo circuito impreso, por lo cual existen múltiples dispositivos electrónicos que en la actualidad manejan este estándar. Además, este protocolo contiene un sistema de detección de colisiones, el cual impide que un dispositivo intente transmitir un dato en el bus cuando otro dispositivo está llevando una comunicación en curso.

13.1 Características del protocolo I²C

El protocolo I²C fue una invención de la empresa Philips Semiconductors específicamente para aplicaciones de audio y video. Este estándar se basa en el uso de dos líneas (SDA y SCL) que transportan datos de forma bidireccional entre dos o más circuitos integrados. El nombre I²C proviene del hecho de que el protocolo comunica circuitos integrados en un bus de 2 líneas.

La comunicación entre los dispositivos presenta dos jerarquías: el maestro y el esclavo. El estándar presenta un mecanismo de detección de colisiones en el bus, con lo cual el maestro solo transmitirá un byte cuando detecte que las líneas de comunicación están libres (estado *Idle*). En el protocolo I²C solo existen tres velocidades de reloj para la comunicación:

a. 100 Kbps (modo estándar)
b. 400 Kpbs (modo rápido)
c. 3.4 Mbps (modo de alta velocidad)

Si bien es cierto que el estándar no especifica un número máximo de dispositivos que se pueden conectar al bus, se debe tener cuidado de que la carga total conectada a las líneas SDA y SCL no supere los 400 pF, ya que, de superar este valor, es posible que la comunicación presente errores. En el gráfico 13.1 se observa un esquema de la comunicación I²C entre dos dispositivos.

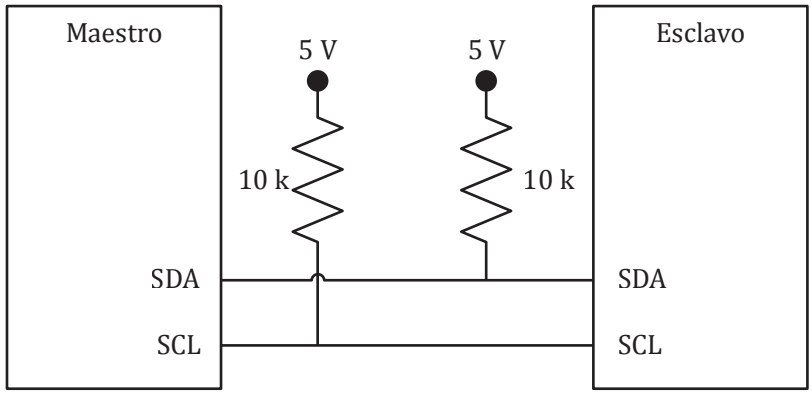
El protocolo I²C simplifica la interfaz de *hardware* entre dispositivos, ya que solo se requieren dos líneas de comunicación entre los periféricos. De esta manera, existen múltiples dispositivos periféricos esclavos que manejan este protocolo y que podrían conectarse al mismo bus. Entre estos dispo-

sitivos se pueden encontrar relojes en tiempo real, acelerómetros, memorias EEPROM, códec de audio, códec de video, entre otros.

En una comunicación I²C puede existir más de un maestro en el bus. Esto significa que dos dispositivos pueden intentar iniciar una comunicación en simultáneo. Para evitar problemas de colisión, el estándar I²C presenta una característica de arbitrariedad que regula que solo un maestro a la vez pueda tomar el control de la bus.

Cuando el maestro intenta transmitir un dato a un dispositivo esclavo, primero debe identificar al dispositivo mediante su dirección única. Cada dispositivo esclavo, de acuerdo con su naturaleza, presenta una dirección, que puede ser de 7 o de 10 bits. De esta manera, cuando el maestro envía una trama inicial con la dirección, esta le llegará a todos los esclavos conectados al bus. Solo aquel dispositivo cuya dirección sea compatible con la enviada por el maestro se mantendrá activo, mientras que los otros dispositivos esclavos estarán deshabilitados.

Gráfico 13.1. Esquema de comunicación I²C entre dos dispositivos



Elaboración propia.

El maestro transmite los 8 bits de una trama a través de la línea SDA. Cada bit en SDA se sincroniza con el pulso de reloj enviado a través de la línea SCL. La señal de reloj siempre es generada por el maestro, pero la línea SDA es bidireccional y la señal de datos puede ser generada tanto por el maestro como por el esclavo. Cuando el maestro requiere la recepción de datos del esclavo, previamente le enviará un comando de lectura para luego generar 8 pulsos de reloj que estarán sincronizados con los datos enviados por el esclavo. Cada bit en la línea SDA se actualiza cuando la línea SCL se encuentra en nivel bajo[58].

Como se puede observar en el gráfico 13.1, el bus I²C utiliza una resistencia de *Pull Up* en cada línea. El valor de la resistencia depende de recomendaciones del fabricante del dispositivo esclavo. Normalmente estos valores se encuentran entre 1 kΩ y 10 kΩ. Estas resistencias son imprescindibles para la comunicación, ya que tanto SDA como SCL se encuentran en configuración colector abierto, tal como se muestra en el gráfico 13.2.

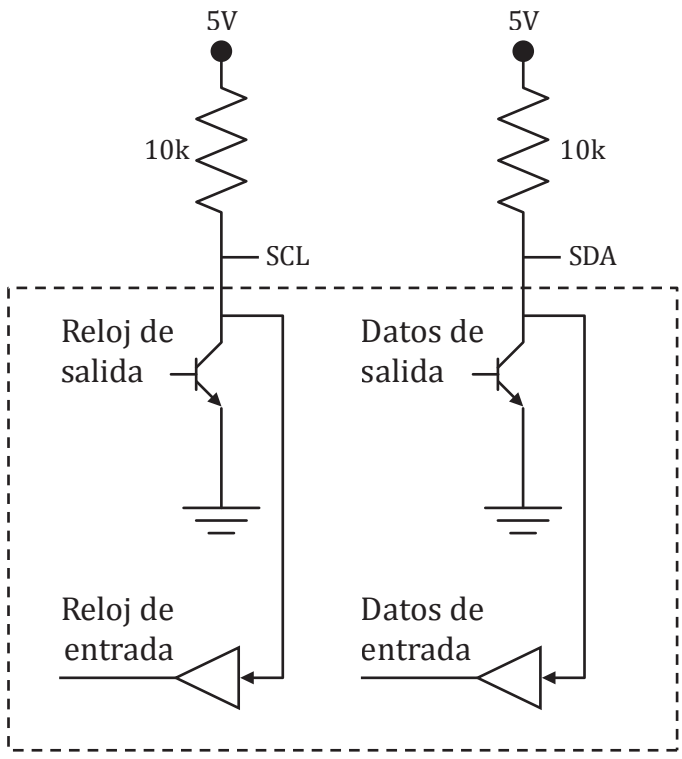
58 Cfr. Cady 2006.

13.2 Transferencia de datos en el protocolo I²C

El protocolo I²C presenta una condición de inicio (START) y otra de parada (STOP) para iniciar y finalizar una transferencia completa de datos, respectivamente. Solo el maestro puede generar las condiciones de START y de STOP. Una vez generada una condición de START, las líneas permanecerán ocupadas, con lo cual ningún otro dispositivo maestro podrá tomar el control del bus hasta que el dispositivo maestro que generó la condición de START culmine la transferencia con una condición de STOP.

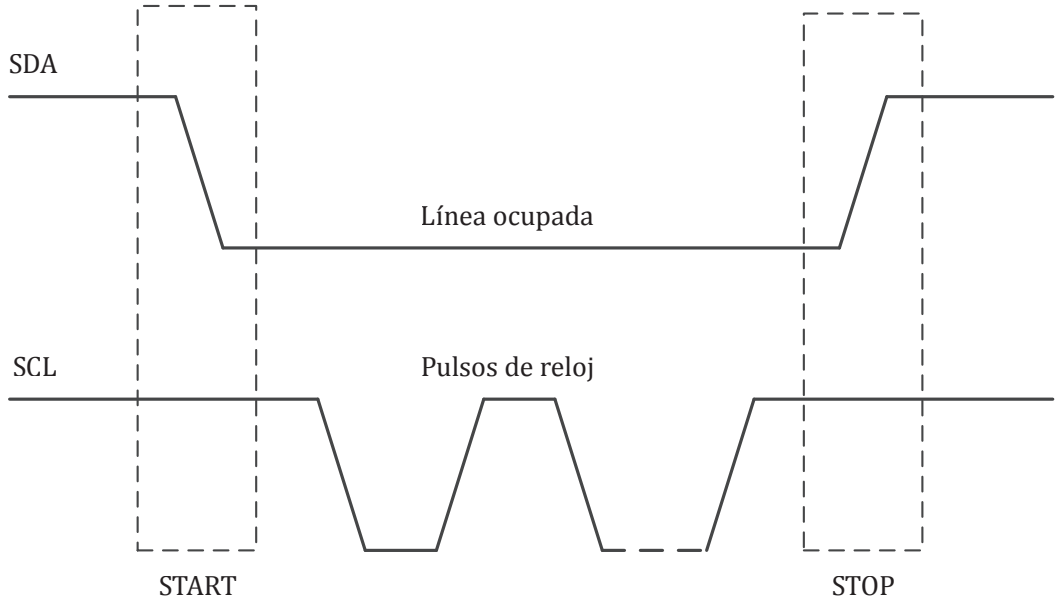
Las condiciones de START y STOP se pueden observar en el gráfico 13.3. En el estado de *Idle* tanto SDA como SCL se mantienen en nivel alto, y es aquí el momento en el cual cualquier maestro puede tomar el control del bus. Cuando se genera la condición de START, la línea SDA pasa a nivel bajo, mientras SCL se mantiene en alto. Después de este momento, SCL puede generar los pulsos de reloj y SDA se actualizará cuando el nivel de la señal de reloj sea bajo. Por otra parte, la condición de STOP se genera cuando la línea SDA pasa a nivel alto, mientras que la señal en SCL se mantiene en nivel alto. Luego de esta condición, ambas líneas permanecen en estado de *Idle*.

Gráfico 13.2. Diagrama interno de los pines del bus I²C para la operación en modo maestro o modo esclavo



Elaboración propia.

Gráfico 13.3. Condiciones de START y STOP en las líneas del bus I²C

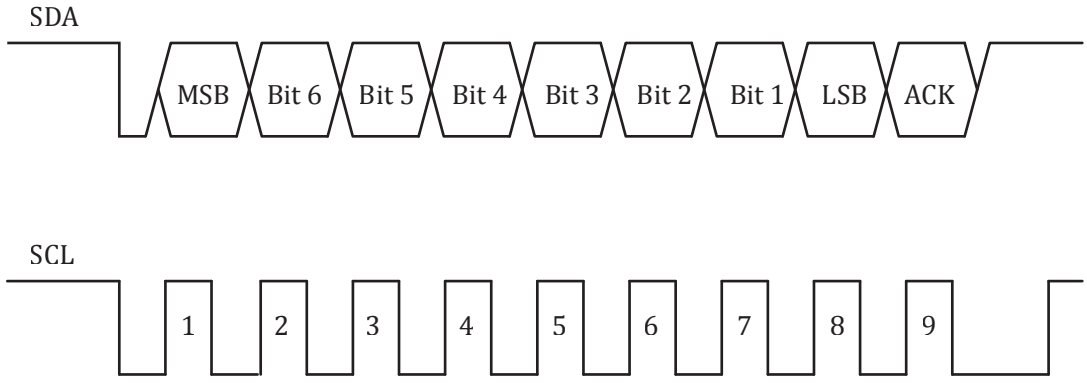


Elaboración propia.

Los pulsos de reloj se componen de 9 periodos por cada byte enviado. Los ocho primeros corresponden a los bits de datos, comenzando por el bit MSB, el cual se transmite al inicio de la trama. Cada uno de estos bits es generado por el maestro. Una vez culminada la transferencia de estos 8 bits, el maestro libera la línea SDA manteniéndose en nivel alto y genera un noveno pulso de reloj. Es en ese momento cuando el esclavo toma la línea SDA y envía un bit '0' sincronizado con el noveno pulso. Este bit se conoce como «bit de reconocimiento» (del inglés *Acknowledge* o ACK), y representa la forma en que el esclavo indica al maestro que acaba de recibir los 8 bits de manera satisfactoria. Una vez detectado este bit, el maestro puede transmitir un nuevo byte, generando nueve ciclos de reloj, o puede liberar la línea con una condición de STOP. Por cada byte recibido, el esclavo siempre deberá generar un bit de ACK. En el gráfico 13.4 se observa un esquema de transmisión de un paquete de 8 bits en el bus I²C. De la misma manera, en un acceso de lectura, el dispositivo esclavo generará los 8 bits de datos y será el maestro el encargado de generar el bit de ACK para confirmar la recepción al dispositivo esclavo.

Podría suceder bajo alguna circunstancia que el dispositivo esclavo no sea capaz de generar el bit de ACK, debido a que se encuentra ocupado al momento de recibir la trama. Ante esta situación, el dispositivo esclavo dejará la línea SDA en nivel alto. Con esto, el maestro detectará que la línea SDA no se mantuvo en nivel bajo en el noveno pulso de reloj y generará una condición interna de NACK (del inglés *No Acknowledge* o no reconocimiento). Ante esta circunstancia, el maestro puede elegir entre liberar el bus con un comando de STOP o generar una condición de reinicio (RESTART). Esta última condición permite que el maestro retome las líneas SDA y SCL para iniciar una nueva transferencia. La condición de NACK también es usada por el dispositivo esclavo para indicar el fin de una transferencia.

Gráfico 13.4. Trama de envío de datos en el bus I²C (8 bits de datos + ACK)



Elaboración propia.

13.3 Arbitrariedad

El protocolo I²C permite que múltiples maestros puedan coexistir en el sistema de comunicación. Cuando dos o más maestros intentan intervenir el bus al mismo tiempo, un esquema de arbitrariedad permite que solo uno de ellos tenga acceso y el resto deba esperar un tiempo adicional.

Los datos en el bus I²C son solo válidos cuando la línea SCL se encuentra en el nivel alto del pulso de reloj. El dispositivo esclavo puede tomar control de la línea SCL y ponerlo a nivel bajo cuando se encuentra procesando un dato, con el fin de darle aviso al maestro de que debe esperar un tiempo adicional para una respuesta.

Cuando dos dispositivos maestros conectados al bus tratan de transmitir un dato, antes deben verificar la condición de la línea SDA, para evitar así hacerlo en simultáneo. Si uno de los maestros llega a detectar la condición de START, deberá dejar libres las líneas y esperar más tiempo, o, en todo caso, reconfigurarse como dispositivo esclavo.

13.4 Direccionamiento de periféricos

Después de que el maestro transmite una condición de START, debe enviar la dirección del dispositivo esclavo para poder ubicarlo en la red de circuitos integrados. Las direcciones a transmitir pueden ser de 7 o de 10 bits. Todos los dispositivos esclavos tienen una dirección predeterminada. Si un sistema embebido es configurado como dispositivo esclavo, su dirección ha de ser asignada por *firmware*, pero el programador deberá tener cuidado de no asignar una dirección similar a la de alguno de los otros dispositivos esclavos conectados en la red.

El primer byte después de la condición de START lleva una dirección de 7 bits y un bit (LSB) que indica el modo de acceso de lectura o escritura (bit R/W). Cuando el bit R/W es '1', el maestro indicará al esclavo que el siguiente acceso deberá ser de lectura. Es decir, la información se transmite de esclavo a maestro. Si el bit R/W es '0', entonces el esclavo deberá esperar por un byte de transmisión del maestro. En el gráfico 13.5 se muestra un esquema del paquete de transmisión de los 7 bits de dirección y el bit R/W. Si el dispositivo esclavo posee una dirección de 10 bits, el maestro deberá enviar los tres

bits restantes de la dirección en un segundo paquete después del bit de ACK generado por el esclavo al recibir los primeros 7 bits.

Gráfico 13.5. Paquete inicial de 7 bits de dirección y el bit de lectura y escritura

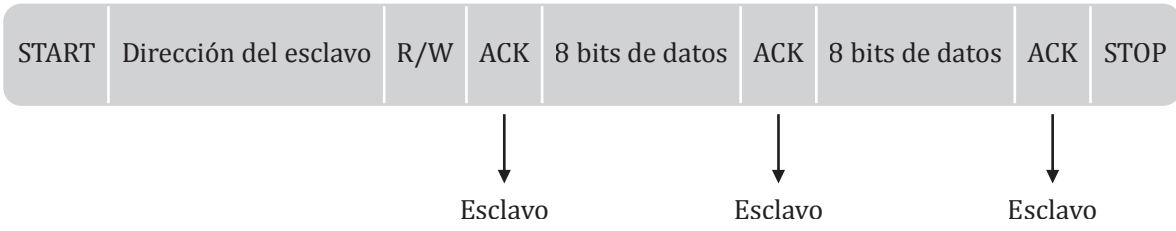
Elaboración propia.

13.5 Formato de transferencia de datos

Existen dos formatos básicos para la transferencia de información en el bus I^2C. El primero corresponde a la transmisión del maestro al esclavo (escritura). El segundo se encuentra representado por la transmisión de datos del esclavo al maestro (lectura). Podría considerarse un tercer formato correspondiente a la combinación de los dos anteriores (normalmente, las transferencias de datos implican accesos de escritura y lectura de manera secuencial).

El formato de transmisión de maestro a esclavo se muestra en el gráfico 13.6. El maestro inicia la comunicación con una condición de START seguida de la transmisión del primer paquete con la dirección de 7 bits (usada para este ejemplo) y el bit R/W en '0' (modo escritura). El esclavo genera inmediatamente una condición ACK llevando la línea SDA a '0'. Una vez detectada esta condición, el maestro puede transmitir la cantidad de tramas de 8 bits que considere conveniente (dependiendo, por supuesto, de la configuración del dispositivo esclavo). Cada trama de 8 bits deberá ser confirmada por un bit de ACK del esclavo para que el maestro pueda transmitir el siguiente paquete. Una vez finalizada la transmisión, el maestro deberá generar la condición de STOP para liberar el bus I^2C.

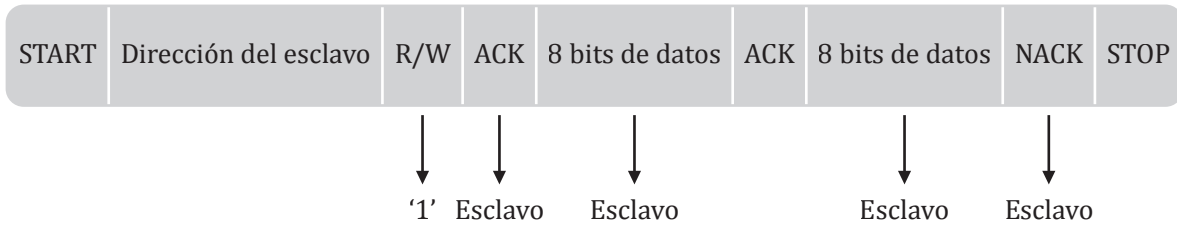
Gráfico 13.6. Formato de transmisión de maestro a esclavo



Elaboración propia.

Por otro lado, en un acceso de lectura la trama de datos es enviada por el dispositivo esclavo, mientras que los bits de ACK son generados por el maestro para confirmar la recepción de los paquetes de 8 bits. En el gráfico 13.7 se muestra este formato de comunicación.

Gráfico 13.7. Formato de transmisión de esclavo a maestro



Elaboración propia.

El maestro inicia la comunicación con una condición de START, seguida de la transmisión de los 7 bits de dirección y el bit de R/W en '1' (modo lectura). El esclavo confirma la recepción de este paquete con una condición ACK y posteriormente comienza a transmitir los 8 bits de datos. La condición de ACK debe ser generada por el maestro al noveno pulso de reloj de datos. Una vez que el dispositivo esclavo no tiene más información para transmitir, confirma el último paquete con una condición de NACK, con lo cual el maestro puede finalizar la comunicación con una condición de STOP.

13.6 Registros de configuración del PIC18F4550

El módulo MSSP del PIC18F4550 visto en el capítulo anterior se puede configurar también en modo I²C. Claro está que al realizar esta configuración queda desactivado el protocolo SPI. Una vez que se configura el MSSP en modo I²C, es posible controlar dispositivos esclavos de 7 y 10 bits de dirección. Los pines usados para la comunicación son RB0 (SDA) y RB1 (SCL) del Puerto B. El registro TRISB se debe configurar como salida digital para que estos pines operen antes de configurar los registros del módulo MSSP.

Adicionalmente, el módulo MSSP puede generar interrupciones para indicar la condición de START, de STOP y para determinar si el bus de encuentra libre. En el gráfico 13.8 se muestra el diagrama de bloques del módulo MSSP configurado para operar en modo I²C.

En el diagrama de bloques se puede observar nuevamente que la relación entre los registros SSPSR y SSPBUF sigue manteniendo la relación de doble buffer en la cual el valor de escritura sobre SSBPUF será transmitido por la línea SDA y el valor leído representará el byte recibido a través de SDA. La configuración de la velocidad de reloj se logra mediante el valor almacenado en SSPADD para el modo maestro. En modo esclavo, el registro SSPADD es utilizado para fijar el código binario de la dirección de identificación de dispositivo esclavo.

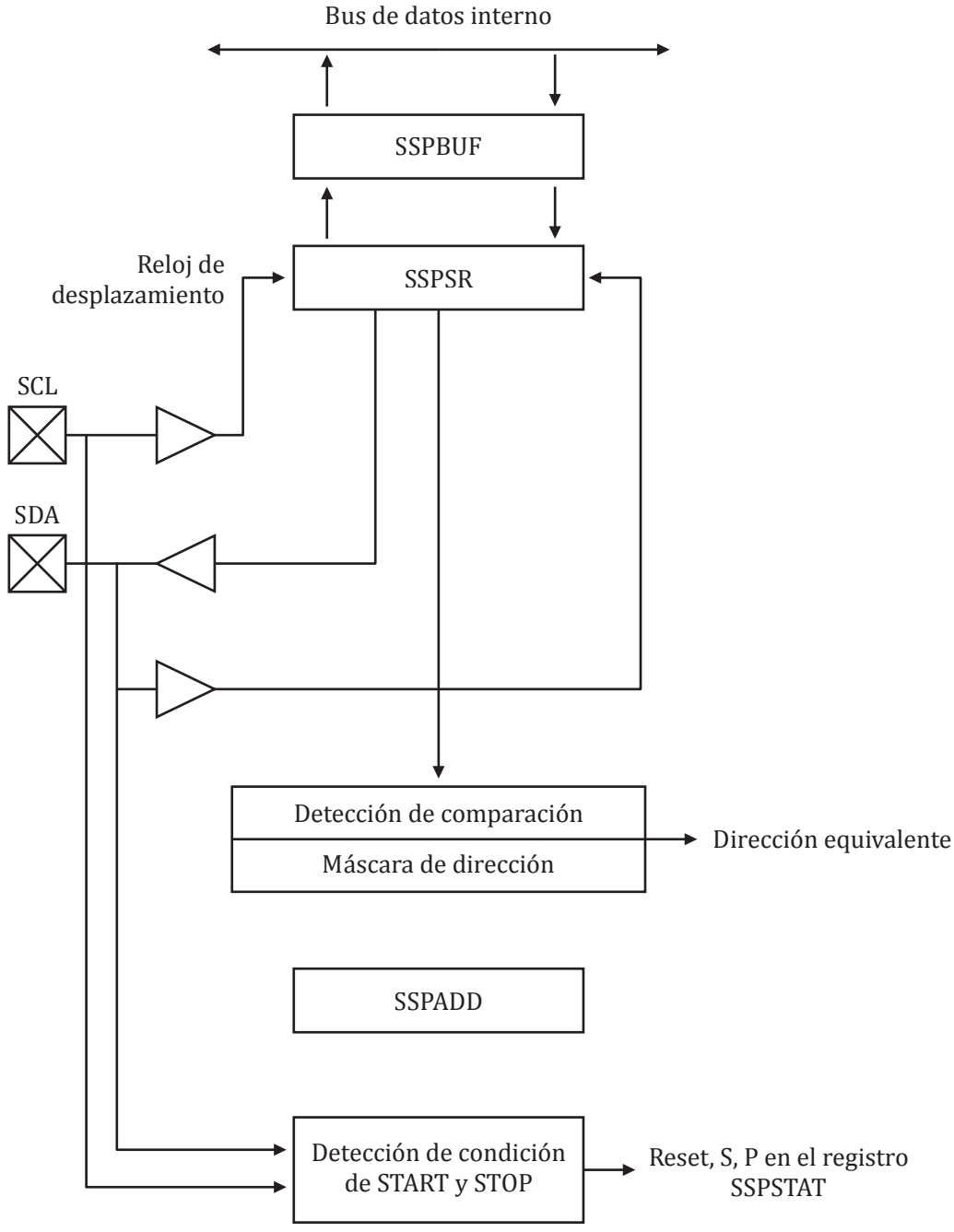
El módulo MSSP posee seis registros de configuración y control:

1. Registro de estado SSPSTAT
2. Registro de control 1 SSPCON1
3. Registro de control 2 SSPCON2
4. Registro buffer SSPBUF
5. Registro de desplazamiento SSPSR (no accesible de manera directa)
6. Registro de dirección o generador de velocidad SSPADD

El registro SSPSTAT se muestra en el gráfico 13.9. Este registro permite configurar el tipo de velocidad. Los 6 bits menos significativos son solo de lectura y permiten verificar el estado de ciertas condiciones de la comunicación. La función de los bits es la siguiente:

El bit SMP permite configurar el modo de velocidad en el cual va a operar el bus I^2C. Si este bit es '1', se selecciona el modo de velocidad estándar a 100 kHz de frecuencia de reloj. En el caso de que este bit se configure en '0' lógico, se seleccionará el modo de alta velocidad a 400 kHz.

Gráfico 13.8. Diagrama de bloques del módulo MSSP en configuración I^2C



Elaboración propia.

Gráfico 13.9. El registro de configuración y estado SSPSTAT

SMP	CKE	D/A	P	S	R/W	UA	BF

Elaboración propia.

El bit CKE permite configurar el bus para operar en modo I²C si CKE es '1'. En caso contrario, el modo del bus queda desactivado.

El bit D/A es de lectura y solo sirve si el sistema embebido se encuentra configurado como esclavo. Si este bit es '1', la última trama de 8 bits recibida corresponderá a un dato. Si el bit es '0', el byte recibido corresponderá a la dirección del esclavo junto con el bit de acceso a lectura o escritura.

El bit P es un indicador de la condición de STOP en el bus. Si este bit es '1', entonces se ha producido una condición de STOP.

El bit S es el indicador de la condición de START en el bus. Si este bit es '1', entonces se ha producido una condición de START.

El bit R/W tiene doble función. En modo esclavo, este bit permite conocer si el acceso del maestro es de lectura ('1') o de escritura ('0'). Cuando el PIC18F4550 se encuentra configurado como maestro, el bit R/W permite conocer si una transferencia se está llevando a cabo. Si el bit R/W se encuentra en nivel '1', indicará que la transmisión se encuentra en progreso. En caso contrario, la transmisión ha sido finalizada.

El bit UA permite saber el momento en el cual se debe actualizar el valor del registro SSPADD cuando el microcontrolador se encuentra configurado como esclavo y con dirección a 10 bits. Esto ocurre cuando el bit UA es '1'.

Finalmente, el bit BF permite saber si el registro SSPBUF se encuentra cargado con un valor y por ende existe una transmisión en curso (BF = '1') o si se ha recibido un dato desde el dispositivo externo (BF = '1'). Este bit opera tanto para el modo recepción como transmisión.

Por otro lado, el registro SSPCON1 es el primer registro de control que debe ser inicializado desde un comienzo para configurar el modo I²C del módulo MSSP y a la vez seleccionar algunas características iniciales. En el gráfico 13.10 se muestra la imagen de este registro. A continuación, se detalla la función de cada uno de sus bits.

Gráfico 13.10. El registro de configuración SSPCON1

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0

Elaboración propia.

El bit de estado WCOL permite detectar una posible colisión en el bus. Si este bit se encuentra en nivel '1', las condiciones del bus no serán las apropiadas para la transmisión de paquetes. Si este bit se encuentra a nivel '0', entonces será posible realizar una transferencia en el bus.

El bit de estado SSPOV se pone a nivel '1' para indicar que se ha recibido un nuevo byte en el registro SSPBUF y el dato previo almacenado en este registro no ha sido leído, por lo cual, este valor anterior se ha perdido.

El bit SSPEN permite activar el bus I²C para su operación. Con un nivel '1', se encenderá el módulo.

El bit CKP solo tiene uso cuando el microcontrolador se encuentra configurado en modo esclavo. Este bit permite realizar el proceso de estrechamiento de reloj, el cual es necesario cuando el dispositivo esclavo requiere más tiempo para procesar un dato y generar el bit ACK. Cuando el bit CKP es '1', la señal de reloj en SCL se libera. En caso contrario, se mantendrá la señal en nivel '0' hasta que el dispositivo esclavo responda con el bit ACK.

Los bits SSPM3:SSPM0 se vieron en el capítulo anterior, cuando se mostró también el registro SSP1CON aplicado a la configuración del bus SPI. Estos bits permiten configurar el modo de operación del bus I²C de acuerdo con las opciones mostradas en el gráfico 13.11.

Gráfico 13.11. Opciones de configuración de los bits SSPM3:SSPM0 del registro SSP1CON

SSMP3	SSPM2	SSPM1	SSPM0	Configuración
0	1	1	0	Modo esclavo con dirección de 7 bits.
0	1	1	1	Modo esclavo con dirección de 10 bits.
1	0	0	0	Modo maestro.
1	0	1	1	Modo maestro controlado por Firmware.
1	1	1	0	Modo esclavo, con dirección de 7 bits e interrupciones activas por condición de START y STOP.
1	1	1	1	Modo esclavo, con dirección de 10 bits e interrupciones activas por condición de START y STOP.

Elaboración propia.

Posteriormente, el registro SSPCON2 permite completar las configuraciones sobre el bus I²C. La función de cada bit de este registro varía de acuerdo con si el modo de operación es maestro o esclavo. En el gráfico 13.12 se observa una imagen de este registro. A continuación, se especifica la función de cada bit tanto para el modo maestro como para el modo esclavo.

Gráfico 13.12. El registro de configuración SSPCON2 en modo maestro

Elaboración propia.

El bit GCEN solo opera para el modo esclavo. Este bit permite habilitar la interrupción cuando se recibe el byte 0x00 en SSPBUF (este valor corresponde a la dirección general o que debe afectar a todos los dispositivos esclavos conectados al bus).

El bit ACKSTAT es un bit de estado que solo opera en modo maestro. Permite saber si el dispositivo esclavo ha generado un bit ACK luego de recibir una trama de 8 bits. El valor de '1' indica que no se ha recibido el bit ACK, mientras que el '0' indica la correcta detección de este bit.

El bit ACKDT en modo maestro permite verificar la condición de ACK que debe ser detectada por el dispositivo esclavo. Cuando este bit es '0', se realiza la condición. En modo esclavo, este bit se transforma en ADMSK5.

El bit ACKEN permite generar una condición de ACK cuando el maestro se encuentra en modo de recepción de datos. Cuando este bit es '1', se inicia la secuencia de ACK en las líneas SDA y SCL. En modo esclavo, este bit se transforma en ADMSK4.

El bit RCEN en modo maestro permite habilitar el modo de recepción de datos, configurando la línea SDA como entrada. Si este bit es '1', se habilita el modo de recepción. En modo esclavo, este bit se transforma en ADMSK3.

El bit PEN permite generar una condición de STOP en las líneas SDA y SCL. Si el bit se pone a '1', esta condición quedará iniciada. En modo esclavo, este bit se transforma en ADMSK2.

El bit RSEN en modo maestro permite generar una condición de RSTART en las líneas SDA y SCL cuando se pone a nivel '1'. En modo esclavo, este bit se transforma en ADMSK1.

Finalmente, el bit SEN permite generar la condición de START en las líneas SDA y SCL cuando se pone a nivel '1' lógico y el módulo MSSP se configura en modo maestro. En modo esclavo, este bit habilita el proceso de estrechamiento de reloj en transmisión y recepción.

Los bits ADMSK5 y ADMSK2 en modo esclavo permiten habilitar el enmascaramiento de los bits del mismo orden en el registro de direcciones SSPADD. El bit ADMSK1 permite habilitar el enmascaramiento del modo de direcciones de 10 bits o 7 bits, según sea la configuración.

La bandera SSPIF del registro PIR1 se puede también poner a nivel '1' lógico cada vez que ocurren las siguientes condiciones: START, STOP, una transferencia de datos por transmisión o recepción, transmisión del ACK y un RSTART.

13.7 Programación del módulo MSSP en modo I²C

Como ya se vio, en el registro SSPCON1 se pueden configurar las siguientes condiciones iniciales para el bus I²C en el microcontrolador:

1. Modo maestro.
2. Modo esclavo a 7 bits.
3. Modo esclavo a 10 bits.
4. Modo esclavo a 7 bits de dirección con interrupción por condición de START y STOP.
5. Modo esclavo a 10 bits de dirección con interrupción por condición de START y STOP.
6. Modo maestro controlador por *firmware*.

Sea cual fuere la configuración, los pines SDA y SCL siempre se mantienen en la condición de Drenador abierto, con lo cual requieren obligatoriamente una resistencia de *Pull Up* externa para poder operar. Estos pines se configuran inicialmente como salidas para modo maestro y entradas digitales para el modo esclavo.

Para la configuración del modo maestro, existen seis opciones de operación posibles:

1. Iniciar una condición de START en SDA y SCL.
2. Iniciar una condición de RSTART en SDA y SCL.
3. Escribir en SSPBUF un byte iniciando una transferencia.
4. Configurar el puerto para la recepción de datos.
5. Generar un ACK al final de la recepción de un byte.
6. Generar una condición de STOP en SDA y SCL.

El maestro genera en todos los casos la señal de reloj en la línea SCL junto con las condiciones de START y STOP. Una transferencia culmina con la condición de STOP o un RSTART, pero en la última situación el bus no será liberado.

El primer byte enviado después del START es la dirección del dispositivo esclavo de 7 u 10 bits seguido del bit R/W de acceso a lectura o escritura que siempre será '0' la primera vez que se transmite. Por cada byte transmitido se produce un pulso de reloj adicional para el bit ACK que deberá ser generado por el dispositivo esclavo.

Cuando el maestro opera en modo de recepción, previamente deberá haber enviado la dirección del dispositivo esclavo seguido de un bit R/W en nivel '1'. Luego, el dispositivo esclavo transmitirá los datos por SDA sincronizado con la señal de reloj en SCL. Una vez terminada la transferencia, el maestro deberá generar el bit ACK.

Una secuencia de transmisión se inicia de la siguiente manera en el dispositivo maestro:

1. Se genera la condición de START con el bit SEN = '1'.
2. Se carga el registro SSPBUF con la dirección del dispositivo esclavo que se desea comunicar.
3. La dirección se desplaza por el pin SDA hasta transmitir los 8 bits.
4. El MSSP lee el bit ACK del esclavo y escribe su valor en el bit ACKSTAT.
5. El MSSP genera una interrupción al final del noveno bit, haciendo que SSPIF = '1'.
6. El usuario carga el registro SSPBUF con 8 bits de datos.
7. Los datos se desplazan por SDA hasta completar los 8 bits.
8. Se lee el bit ACK y este estado se detecta leyendo el bit ACKSTAT.
9. El módulo MSSP habilita la bandera SSPIF con '1' al final del noveno bit.
10. El usuario genera una condición de STOP con el bit PEN en nivel '1'.

Para configurar la tasa de bit del reloj SCL en modo maestro, se debe utilizar la ecuación que se muestra a continuación:

$$Tasa\ de\ bit = \frac{F_{OSC}}{4(SSPADD + 1)}$$

Donde la tasa de bit puede ser de 100 kHz, 400 kHz o 1 MHz. F_{OSC} es la frecuencia de reloj y SSPADD es un registro de 8 bits. Por ejemplo, para una frecuencia de reloj de 48 MHz y una velocidad deseada de 400 kHz, SSPADD deberá tomar el valor de 29.

El compilador MPLAB XC8 cuenta con un conjunto de librerías específicas para el manejo de diversos periféricos incluidos en el módulo MSSP. Para el control del bus I²C se cuenta con la librería i2c.h, que se encuentra en la carpeta *Include*/*Plib* generada por la instalación del compilador. Esta librería contiene una serie de funciones que permiten manejar todas las opciones del modo I²C. Estas funciones se muestran en el gráfico 13.13.

Gráfico 13.13. Funciones de la librería i2c.h

Función	Descripción
AckI2C()	Permite generar una condición de ACK del maestro al dispositivo esclavo.
CloseI2C()	Desactiva el módulo MSSP.
DataRdyI2C()	Permite determinar si existe un dato disponible en SSPBUF.
getcI2C()	Retorna el byte enviado por el dispositivo esclavo.
getsI2C()	Retorna una cadena de bytes enviados por el dispositivo esclavo.
IdleI2C()	Realiza una espera hasta que el bus retorne al estado de Idle.
NotAckI2C()	Genera una condición de NACK en el bus I²C.
OpenI2C(unsigned char sync_mode, unsigned char slew)	Permite configurar el módulo MSSP para operar en modo I²C. El primer parámetro puede ser SLAVE_7, SLAVE_10 o MASTER (esclavo a 7 bits de dirección, 10 bits de dirección o modo maestro respectivamente). El segundo parámetro puede ser SLEW_OFF o SLEW_ON (modo de 100kHz habilitado o modo de 400 kHz habilitado, respectivamente).
putcI2C(unsigned char num)	Transmite un byte al dispositivo esclavo.
putsI2C(unsigned char *num)	Transmite una cadena de bytes al dispositivo esclavo.
ReadI2C()	Permite leer un solo byte del bus I²C.
RestartI2C()	Genera una condición de RSTART en el bus I²C.
StartI2C()	Genera una condición de START en el bus I²C.
StopI2C()	Genera una condición de STOP en el bus I²C.
WriteI2C()	Escribe un byte en el bus I²C.

Elaboración propia.

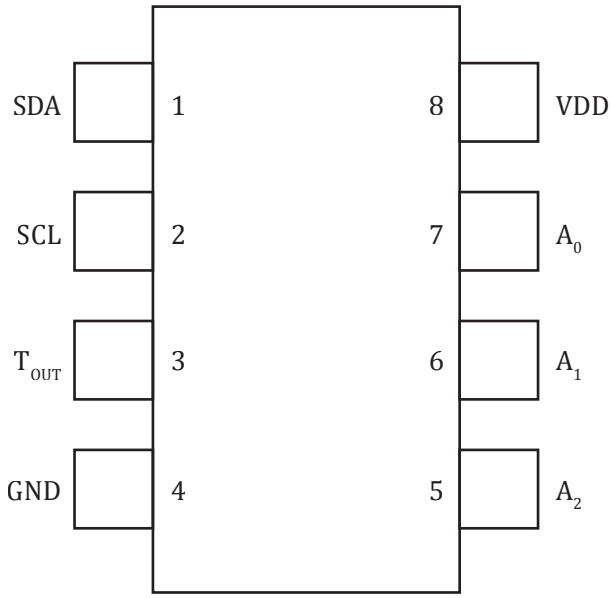
13.8 Ejemplos de aplicación con el módulo I²C del PIC18F4550

Para comprender de manera práctica la operación del módulo MSSP en modo I²C, se presentan dos ejemplos aplicados. El primero permite monitorizar la temperatura ambiente en base a un termostato en empaque de circuito integrado como lo es el DS1621, que posee un protocolo I²C para el envío de la temperatura. El segundo ejemplo se basa en la lectura y escritura de una memoria EERPOM 24LC08 que, de la misma manera, cuenta con un protocolo I²C.

13.8.1 Lectura de la temperatura usando el sensor DS1621

El sensor de temperatura DS1621 de la empresa Maxim Dallas permite obtener la temperatura ambiente con una precisión de 9 bits. Este sensor tiene una presentación de circuito integrado en empaque DIL 8 (montaje superficial), tal como se muestra en el gráfico 13.14.

Gráfico 13.14. Diagrama de pines del circuito integrado DS1621



Elaboración propia.

El circuito integrado se alimenta con 5 V de tensión y presenta tres pines de direccionamiento (A_0, A_1 y A_2) para poder conectar hasta 8 sensores en el mismo bus I²C controlados por el mismo maestro. El pin T_{OUT} genera un nivel '1' cada vez que la temperatura medida supera un valor umbral configurado digitalmente por el maestro. Por otro lado, el bus I²C del sensor puede operar tanto en modo de 100 kHz como en 400 kHz de frecuencia.

El valor de temperatura obtenido se presenta en formato complemento a 2 gracias al comando *ReadTemperature*. El rango de medición del sensor es de -55 °C a +125 °C en incrementos de 0.5 °C. El DS1621 puede ser programado en modo de monitoreo continuo o de una única conversión. Estas

opciones se manejan a través del registro de configuración del sensor, el cual se muestra en el gráfico 13.15. La función de cada bit se explica a continuación.

Gráfico 13.15. Registro de configuración del sensor DS1621

DONE	THF	TLF	NVB	X	X	POL	1 SHOT

Elaboración propia.

El bit DONE es el indicador de una conversión de temperatura completa. Cuando este bit se encuentra en nivel '0', la conversión se está llevando a cabo. Si el bit es '1', la muestra de temperatura se encontrará lista para ser leída por el dispositivo maestro.

El bit THF es el bit de bandera de temperatura superior a un valor umbral que se carga en el registro TH del sensor.

El bit TLF es el bit de bandera de temperatura menor o igual al valor umbral que se carga en el registro TL del sensor.

El bit NVB es la bandera de la memoria EEPROM del sensor ocupada. Si este bit se encuentra en nivel '1', la memoria EEPROM del sensor estará siendo accedida y no podrá ser utilizada por el maestro.

El bit POL permite configurar la polaridad del dato enviado por el bus I²C. Con un nivel '1', se activará el cambio de polaridad.

El bit 1 SHOT permite establecer el modo de conversión continua (nivel '0') o el modo de una única conversión (nivel '1') después del comando *START CONVERT T*.

Cada registro del sensor DS1621 posee un código binario de acceso a memoria. En el gráfico 13.16 se enlistan los comandos de acceso y la función que cumple cada uno de estos. Cada comando debe ser transmitido por el maestro en el bus I²C.

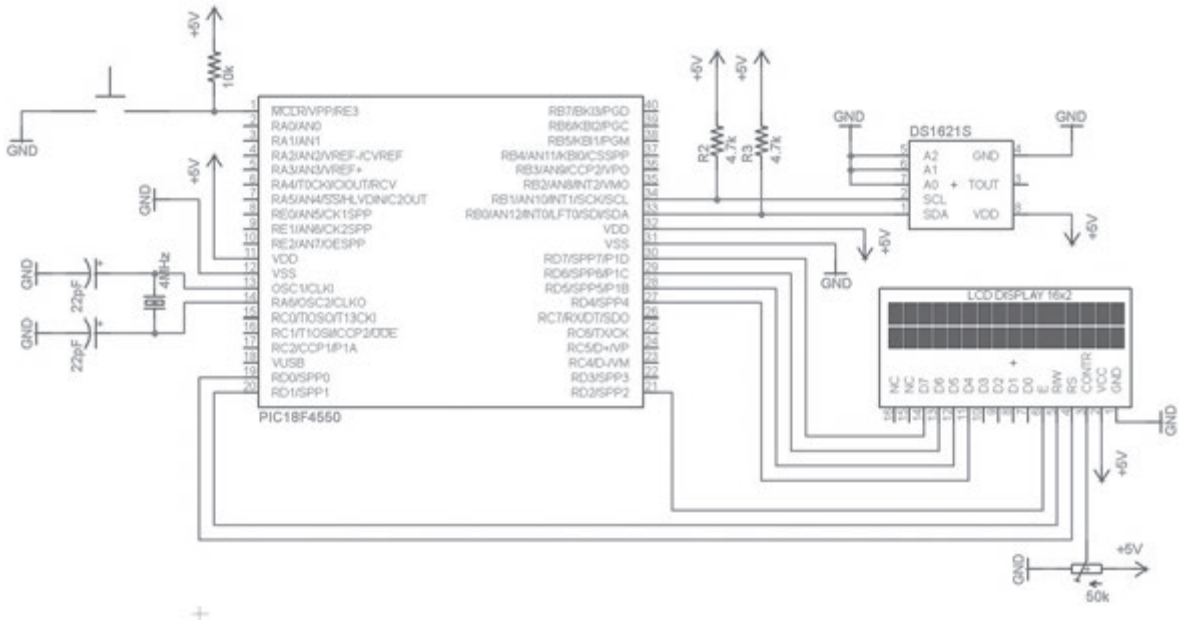
Gráfico 13.16. Lista de comandos para la configuración del sensor de temperatura DS1621

Comando	Código	Descripción
ReadTemperature	0xAA	Este comando solicita la lectura de un valor de temperatura.
Access TH	0xA1	Este comando solicita el acceso a la configuración del umbral de temperatura alto para el funcionamiento de la alarma.
Access TL	0xA2	Este comando solicita el acceso a la configuración del umbral de temperatura bajo para el funcionamiento de la alarma.
Start Convert	0xEE	Este comando inicia una conversión que puede ser única o en modo progresivo.
Stop Convert	0x22	Este comando detiene el proceso de conversión de temperatura.
Access Config	0xAC	Este comando permite acceder al registro de configuración del sensor.

Elaboración propia.

En el gráfico 13.17 se observa el circuito esquemático propuesto para la adquisición de temperatura usando el sensor DS1621 y el PIC18F4550. Las resistencias de *Pull Up* del bus I^2C se fijan en 4.7 kΩ por recomendación del fabricante del sensor. Las líneas A_2, A_1 y A_0 del DS1621 se conectan a tierra, con lo cual los tres últimos bits de su dirección serán 000_2. La dirección completa de 7 bits entonces resultará ser 1001000_2. El módulo LCD se conecta al Puerto D del microcontrolador para hacer uso de la librería LCD, a fin de mostrar el valor de temperatura en la pantalla.

Gráfico 13.17. Circuito esquemático del sistema de adquisición de temperatura mediante el sensor DS1621



Elaboración propia.

El valor de temperatura enviado por el sensor se compone de dos bytes: byte MSB y el byte LSB. El primer byte contiene el valor de temperatura en formato complemento a 2. El segundo byte solo utiliza su bit más significativo para indicar si la temperatura presenta una fracción de 0.5 °C (nivel '1') o no (nivel '0'). Por ejemplo, la cifra 011110100000000_2 representa la temperatura de 125 °C. Por otro lado, el valor binario 1110011110000000_2 representa -25.5 °C.

El acceso al sensor requiere de un maestro que genere la condición de START seguido de la dirección del esclavo y el bit de lectura escritura en nivel 0_2. Posteriormente, se debe esperar la condición de ACK del sensor y luego enviar la dirección del registro de configuración (0xAC), para luego transmitir el valor de configuración, el cual se elige 0x02 para obtener un modo de conversión continua con polaridad activa. Una vez realizada esta operación, se deben esperar 11 milisegundos aproximadamente para establecer un nuevo comando, ya que se estaría realizando una escritura sobre la memoria EEPROM del sensor.

Una vez realizada la inicialización del sensor, el maestro debe transmitir el comando de inicio de conversión (0xEE) seguido de la solicitud de acceso al valor de temperatura (0xAA). Luego de esto, el sensor transmitirá dos bytes con la información de temperatura. Una vez recibidos los bytes, el DS1621 generará una condición de NACK y finalmente el maestro terminará generando la condición de STOP. Este proceso se deberá repetir cada vez que se requiera una nueva lectura de temperatura.

En el programa 13.1 se muestra un ejemplo de adquisición de temperatura cada 400 milisegundos, aproximadamente. El valor de temperatura se muestra en la pantalla LCD con una precisión de 1 °C (se ignora el segundo byte de la trama de temperatura que contiene el factor decimal). El valor obtenido se transforma en su complemento a 2 en caso de que sea negativo (si el bit MSB es '1') y ante esta situación se agrega el símbolo ASCII '-' (negativo) en la pantalla LCD antes de enviar el valor numérico de la temperatura.

Programa 13.1. Rutina de adquisición de temperatura del sensor DS1621 a través del bus I²C

```
#include <xc.h>
#include <plib/i2c.h>
#include <plib/delays.h>

#pragma config FOSC   = XTPLL_XT       //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV = 1              //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV = OSC1_PLL2      //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT    = ON
#pragma config WDT     = OFF
#pragma config LVP     = OFF
#pragma config BOR     = OFF
#pragma config DEBUG   = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE   = ON

#include "LCD.h"

char _byte_MSB, _byte_LSB;
void Init_DS1621(void);
int Lee_DS1621(void);
int temp;
void main()
{
        PORTB = 0;
        TRISD = 0x00;
        TRISB = 0x7C;
        LATD = 0;
        Delay1KTCYx(180);              //Retardo de 15 mseg
        LCD_CONFIG();
        OpenI2C(MASTER, SLEW_ON);  // Inicializa el módulo I2C.
        SSPADD = 29;                   //400kHz para 48MHz de reloj.
```

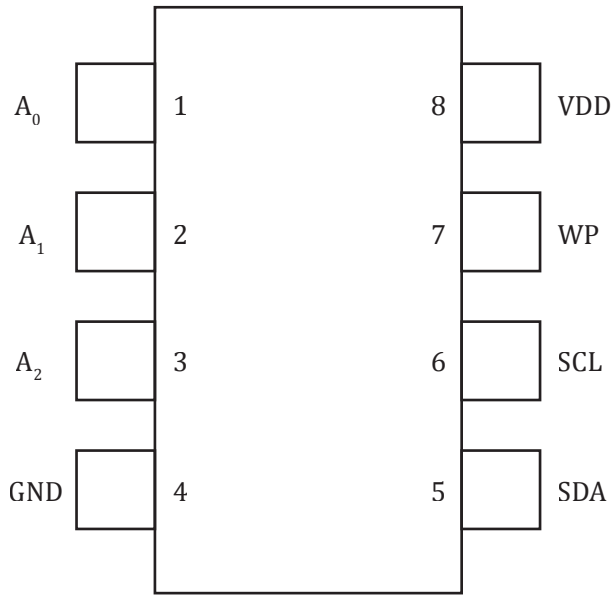
```
        Init_DS1621();                  //0xA0 es la dirección del dispositivo esclavo.
        temp = Lee_DS1621();
        ESCRIBE_MENSAJE("Temp:",5);
        CURSOR_ONOFF(OFF);
        while(1)
        {
                POS_CURSOR(1,5);
                if(temp>= 0x8000)
                {
                        ENVIA_CHAR('-');
                        temp-=0x8000;
                }
                ENVIA_CHAR(temp/100+'0');
                ENVIA_CHAR((temp%100)/10+'0');
                ENVIA_CHAR((temp%100)%10+'0');
                ENVIA_CHAR('C');
                ENVIA_CHAR(' ');
                Delay10KTCYx(240);
                Delay10KTCYx(240);
                temp = Lee_DS1621();
        }
}
void Init_DS1621(void)
{
        IdleI2C();
        StartI2C();             //Condición de START.
        WriteI2C(0x90);         //Envío de la dirección del dispositivo esclavo para acceso de
                                //escritura.
        WriteI2C(0xAC);         //Acceso al registro de configuración.
        WriteI2C(0x02);         //Byte de configuración para conversión continua y alta
                                //polaridad.
        StopI2C();              //Condición de STOP
        Delay1KTCYx(132);       //Retardo de 11 milisegundos
}
int Lee_DS1621(void)
{
        int valor;
        char num_MSB,num_LSB;
        //Inicio de la conversion de temperatura
```

```
        RestartI2C();           //Condición de reinicio.
        WriteI2C(0x90);         //Se envía nuevamente la dirección del dispositivo esclavo
                                //para acceso a escritura.
        WriteI2C(0xEE);         //Se transmite el comando de inicio de conversión.
        StopI2C();              //Condición de STOP.

        StartI2C();             //Condición de START.
        WriteI2C(0x90);         //Envío de la dirección del dispositivo esclavo para acceso a
                                //escritura.
        WriteI2C(0xAA);         //Acceso al registro de temperatura.
        RestartI2C();           //Condición de RSTART.
        WriteI2C(0x91);         //Envío de la dirección del dispositivo esclavo para
                                //acceso a lectura.
        num_MSB = ReadI2C();    // Lectura del byte MSB de temperatura.
        AckI2C();               // Envío de la condición de ACK del maestro.
        num_LSB = ReadI2C();    // Lectura del byte LSB de temperatura.
        NotAckI2C();            // Envío de la condición NACK del maestro.
        StopI2C();              //Condición de STOP.
        if(num_MSB>=0)
        {
                valor = num_MSB;
                return(valor);
        }
        else
        {
                num_MSB = ~num_MSB+1; //Complemento a 2.
                valor = 0x8000 + num_MSB;
                return(valor);
        }
}
```

13.8.2 Interfaz con una memoria EEPROM 24LC08

La memoria EEPROM 24LC08 es un circuito integrado fabricado por la empresa Microchip que presenta un bus de comunicación I²C para el acceso a lectura y escritura. La capacidad total de esta memoria es de 1 kB. La velocidad de acceso se puede configurar tanto para el modo de 100 kHz como para 400 kHz, y la tensión de alimentación de la memoria puede alternar entre 2.5 y 5.5 V. En el gráfico 13.18 se observa el diagrama de pines de la memoria.

Gráfico 13.18. Diagrama de pines de la memoria EEPROM 24LC08



Elaboración propia.

Al igual que el sensor de temperatura, la memoria EEPROM posee tres pines de dirección A_0, A_1 y A_2 para modificar la dirección de hasta 8 memorias idénticas conectadas al bus I^2C. El pin WP (protección contra escritura, del inglés *Write Protection*) permite habilitar el proceso de escritura en la memoria (nivel '0') o deshabilitarlo con un nivel '1'. La dirección esclava de 7 bits de la memoria EEPROM es 0xA0 (donde los bits A_0, A_1 y A_2 son los tres bits LSB de la dirección).

La librería i2c.h que provee Microchip en el compilador MPLAB XC8 contiene funciones preparadas para el control de toda la familia de memorias EEPROM de Microchip 24LCXX. En total se cuenta con un total de cinco funciones para el acceso a lectura y escritura de la EEPROM. En el gráfico 13.19 se enumeran todas estas funciones.

Gráfico 13.19. Funciones para el manejo de la memoria EEPROM 24LC08

Función	Descripción
EEAckPolling(unsgined char control)	Esta función permite establecer la correcta ubicación de la memoria EEPROM en el bus. Se retorna el valor de 0x00 si es que no hay problemas en la comunicación.
EEByteWrite(unsigned char control, unsigned char address, unsigned char data)	Esta función permite accede al dispositivo mediante el comando de control (dirección del esclavo y acceso a lectura) y guardar en la dirección especificada en *address* el valor especificado en *data.*
EECurrentAddRead(unsigned char control)	Lee el byte especificado por el contador interno de direcciones que automáticamente se incrementa en 0x01 después del acceso a lectura.
EEPageWrite(unsigned char control, unsigned char address, unsigned char *wptr)	Esta función escribe una cadena de caracteres en la memoria EEPROM a partir de la dirección especificada en *address*. La cadena se recibe en el parámetro *wptr.

Función	Descripción
EERandomRead(unsigned char control, unsigned char address)	Esta función retorna el valor almacenado en la dirección especificada en *address*. La palabra de control deberá tener el bit de R/W en nivel 1_2 lógico para el acceso a lectura.
EESequentialRead(unsigned char control, unsigned char address, unsigned char *rdptr, unsigned char length)	Esta función retorna en un buffer especificado en rdptr el conjunto de bytes almacenados en la memoria EEPROM entre las direcciones especificadas por *address* y *address*+ *Length.*

Elaboración propia.

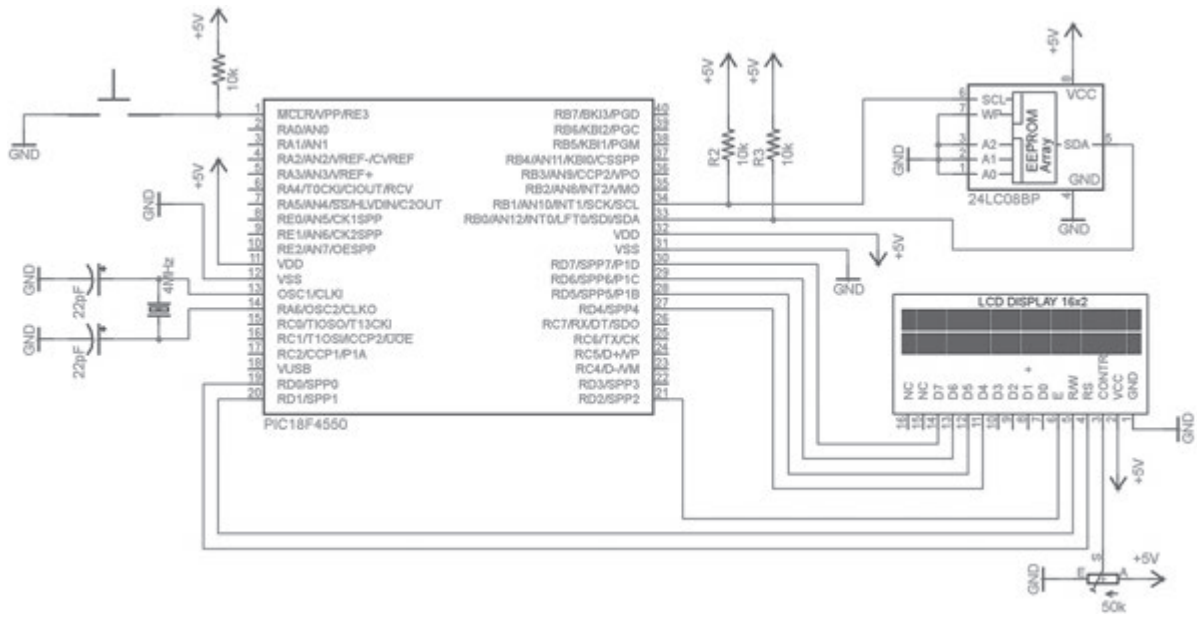
En el gráfico 13.20 se muestra el circuito esquemático para la conexión de la memoria EEPROM 24LC08 y el PIC18F4550. Las resistencias de *Pull Up* se fijan en 10 kΩ por recomendación del fabricante del chip. En el programa 13.2 se muestra un ejemplo de grabación del mensaje "PIC" entre las direcciones de la EEPROM 0x30-0x32. Posteriormente, se realiza la lectura de cada una de estas tres direcciones y se muestra en la pantalla LCD el mensaje.

Programa 13.2. Rutina de acceso a lectura y escritura de la memoria EEPROM 24LC08

```
#include <xc.h>
#include <plib/i2c.h>
#pragma config FOSC   = XTPLL_XT       //XT_PLL_XT Opción de reloj del CPU.
#pragma config PLLDIV = 1              //Divisor del PLL (4MHz/1)*24=96MHz.
#pragma config CPUDIV = OSC1_PLL2      //Divisor del Postscaler 96MHz/2 = 48MHz.
#pragma config PWRT   = ON
#pragma config WDT    = OFF
#pragma config LVP    =OFF
#pragma config BOR    = OFF
#pragma config DEBUG  = ON
#pragma config VREGEN = OFF
#pragma config PBADEN = OFF
#pragma config MCLRE  = ON
#include "LCD.h"
unsigned char val;
void main()
{
        TRISD = 0x00;
        TRISB = 0xFC;
        LATD = 0;
        Delay1KTCYx(180);              //Retardo de 15 mseg
        LCD_CONFIG();
```

```
        OpenI2C(MASTER, SLEW_ON);  //Inicializa el módulo I2C.
        SSPADD = 29;               //400kHz para 48MHz de reloj.
        EEByteWrite(0xA0, 0x30, 'P');  //0xA0 es la dirección.
        EEAckPolling(0xA0);
        EEByteWrite(0xA0, 0x31, 'I');
        EEAckPolling(0xA0);
        EEByteWrite(0xA0, 0x32, 'C');
        EEAckPolling(0xA0);
        val = EERandomRead(0xA0,0x30);
        ENVIA_CHAR(val);
        val = EERandomRead(0xA0,0x31);
        ENVIA_CHAR(val);
        val = EERandomRead(0xA0,0x32);
        ENVIA_CHAR(val);
        while(1);
}
```

Gráfico 13.20. Circuito esquemático para la comunicación entre el PIC18F4550 y la memoria EEPROM



Elaboración propia.

Video

Diseño de un reloj en tiempo real haciendo uso del circuito integrado DS1307 y el bus de comunicación I2C.

http://bit.ly/1L4sIIP

Resolución de videos en HD

Preguntas de repaso

1. ¿Cuál es la función de las resistencias de *Pull up* en las líneas SDA y SCL?
2. ¿Qué función cumple el bit ACK en la comunicación I²C?
3. Explique en qué consiste la arbitrariedad.
4. Si se tienen cuatro dispositivos esclavos DS1621 conectados al puerto I_2C del PIC18F4550, ¿cómo es posible diferenciarlos para un acceso a lectura?
5. ¿En qué se diferencia la condición de START de la condición de RESTART?
6. ¿Cómo se determina la velocidad o frecuencia de reloj del maestro en el módulo MSSP del PIC18F4550?

Respuestas

Respuestas del capítulo 13:

http://bit.ly/1UEwmtK

Capítulo 14. El módulo USB

Una de las características más resaltantes del PIC18F4550 es su módulo USB interno, que permite la transferencia de información al computador a través de dicha interfaz. Con este periférico, el microcontrolador puede realizar diversas aplicaciones, entre las que se encuentran simular el funcionamiento de un ratón, teclado o *joystick*, o también otro dispositivo específico de una clase USB conocida. Esto, obviamente, abre un abanico de posibilidades para realizar múltiples tipos de aplicaciones que requieran comunicación entre un dispositivo terminal y el computador. Además, como ya se vio en el capítulo 11, el puerto RS232 es prácticamente inexistente en las computadoras personales de hoy en día, y una alternativa a futuro para que un *hardware* a diseñar se pueda comunicar sin requerir un protocolo RS232, seguirá siendo por varios años más el puerto USB.

El protocolo USB es complejo y extenso. En este capítulo no se desea ahondar en el protocolo USB, sino más bien mencionar algunas características importantes, para luego comprender de forma general las etapas necesarias para diseñar un *hardware* de interconexión con el PIC18F4550 y un computador a través del bus USB.

El estándar USB nace específicamente en 1996 con el sistema operativo Windows 95 para reemplazar a los protocolos RS232, LPT y PS/2, que eran los más usados en aquellos tiempos en las computadoras personales. Es por esta razón que el protocolo USB es específico para la comunicación de un dispositivo externo con un computador, aunque en los últimos años, con la aparición del estándar USB On the Go (OTG), algunos sistemas embebidos pueden cumplir con la función del computador y conectarse a dispositivos periféricos (por ejemplo, el PIC32MX460F512L de Microchip).

La comunicación USB es de tipo serial asíncrona en modo Half dúplex (hasta la versión 2.0, la versión USB 3.0 es Full dúplex). El computador cumple con la función de maestro o *Host* y toma el control de la comunicación en el bus. Los dispositivos periféricos cumplen con la tarea de responder a los comandos enviados por el *Host* y es por esta razón que se denominan dispositivos esclavos. Este esquema de comunicación produce múltiples beneficios para los usuarios de los productos que manejan este protocolo, entre los cuales se encuentran la fácil configuración de los dispositivos, la conexión de múltiples dispositivos al mismo controlador (hasta 127 como máximo), autoconfiguración y autodetección de los dispositivos conectados al computador, conectores estandarizados y de tamaño reducido, un protocolo robusto de comunicación, es barato y provee energía a los dispositivos (5 V a 500 mA), evitando el uso de fuentes de voltaje externas.

Sin embargo, no todo son beneficios. Algunas desventajas del protocolo USB son la corta distancia del cable de comunicación que une el *Host* con el dispositivo esclavo (no debe superar los 3 metros), el hecho de que se requiera más esfuerzo por parte del desarrollador de *hardware* en crear el *firmware* de comunicación, precisa el uso de un *driver* o *software* de interfaz compatible con el sistema operativo del computador y, si se va a comercializar el producto, se deben pagar anualidades por el derecho de

un código de vendedor (*Vendor ID*) y de un código de producto (*Product ID*) que van especificados en el *firmware* del dispositivo esclavo[59].

De esta manera, para realizar un proyecto de comunicación entre un dispositivo esclavo y el computador a través del puerto USB se deben tomar en cuenta los siguientes elementos y programas, necesarios para el funcionamiento de la interfaz:

Del lado del *hardware*: el tipo de sistema embebido, el controlador USB a utilizar y el *firmware* que maneja la comunicación USB a través del controlador.

Del lado del *Host*: el sistema operativo a emplear, el *driver* de dispositivo y el *software* de aplicación (Visual C# .Net, Visual C++, Eclipse, entre otros).

Todos estos elementos tienen que ser creados o reutilizados en el proyecto por parte del diseñador. En algunos casos, no es necesario crear un *Driver* de dispositivo o un *software* de aplicación, sino más bien tomar aquellos que ya se encuentran disponibles y en funcionamiento en el sistema operativo. De esta manera, se puede reutilizar un *driver* para el manejo de otros periféricos USB y crear un *software* de interfaz que haga uso de estos *drivers* para obtener la comunicación con el sistema embebido, de acuerdo con los requerimientos de la aplicación.

En la actualidad existen dos estándares USB: El USB 3.0 y el USB 2.0. El primero aplica a dispositivos de alta velocidad (hasta 5 Gbps), como sistemas de video o dispositivos de audio, y es uno de los estándares más recientes. El segundo contiene tres tipos de velocidades: *Full Speed* (a 480 Mpbs), *High Speed* (a 12 Mpbs) y *Low Speed* (a 1.5 Mpbs). El PIC18F4550 es compatible con el estándar USB 2.0 con velocidades configurables entre 1.5 Mpbs y 12 Mpbs[60].

14.1 Fundamentos del bus USB

La comunicación USB nace especialmente para la transferencia de información entre un dispositivo esclavo y un computador. Dentro del computador existe un controlador USB denominado *Host*, el cual se comunica con un *Hub* raíz que permite la interconexión de hasta 127 dispositivos periféricos al *Host* (siempre y cuando se conecten más *Hubs* externos al *Hub* raíz, el cual no cuenta con 127 conectores de expansión). En el gráfico 14.1 se muestra un esquema del proceso de conexión de los periféricos al *Host*.

Cada dispositivo conectado al bus USB posee una dirección de 7 bits asignada por el *Host* para poder ser identificado en la red de dispositivos (muy similar a la dirección usada en el bus I^2C). El *Host*, además de información, provee la energía necesaria para que cada dispositivo esclavo pueda operar. Esto incluye una tensión constante de 5 V, que puede entregar hasta un máximo de 100 mA por dispositivo durante el proceso inicial de enumeración y un total de 500 mA cuando el dispositivo se encuentra completamente enumerado en el estándar USB 2.0.

El bus de datos del protocolo USB utiliza dos líneas diferenciales D+ y D-. Junto con estas líneas se encuentran los conductores de alimentación de 5 V y tierra (GND), los cuales proveen la energía a los dispositivos esclavos. Las líneas D+ y D- poseen una tensión variable con referencia a GND (entre 0 y 3.6 V). El código binario se encuentra codificado a través de la diferencia de tensión entre los pines D+ y D-. Un nivel '1' en el bus USB viene a estar representado por una tensión en la línea D+, que es superior a la tensión de la línea D- por un factor de 0.2 V cuando la línea D+ posee un voltaje con referencia

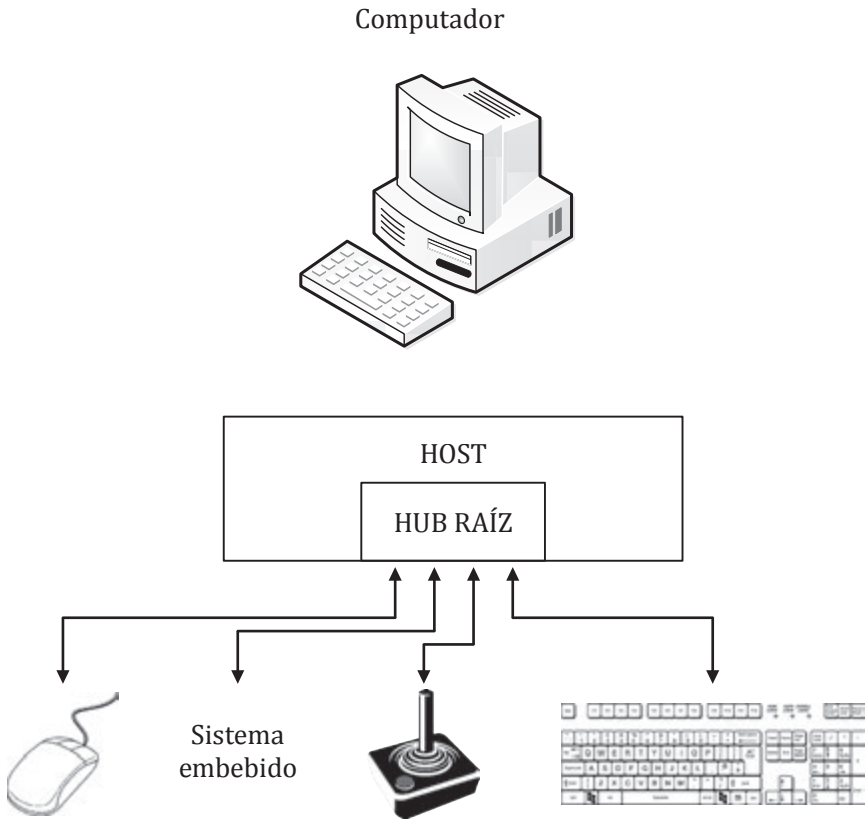
59 Cfr. Axelson 2009.

60 Cfr. Axelson 2009.

a GND mayor que 2 V. Un nivel '0', en cambio, se representa cuando la tensión en la línea D- es mayor que 2 V con referencia a tierra y supera a D+ en 0.2 V. Esto sucede en el caso de las velocidades bajas y medias (*Low* y *Full Speed*). Para la alta velocidad (*High Speed*), la diferencia de tensión entre las líneas del bus debe superar los 0.35 V[61].

El *Hub* raíz contiene resistencias de *Pull down* en los pines D+ y D- con valores variables entre 14.25 kΩ y 24.8 kΩ. Esto implica que ambas líneas presentan un nivel '0' cuando están desconectadas. De esta manera, el *Host* puede conocer la ausencia de un dispositivo conectado al bus. Por el lado del dispositivo esclavo, este presenta una resistencia de *Pull up* entre 900 Ω y 1.5 kΩ de valor conectada a la línea D+ en el caso de la velocidad media y D- para la velocidad baja. De esta manera, el *Host* puede conocer cuál es velocidad esperada del dispositivo esclavo, valiéndose de la polaridad de cada línea. En el diagrama del gráfico 14.2 se pueden observar los dos modos de configuración de las resistencias de *Pull up*.

Gráfico 14.1. Esquema de conexión de los dispositivos esclavos al *Hub* raíz del *Host* USB



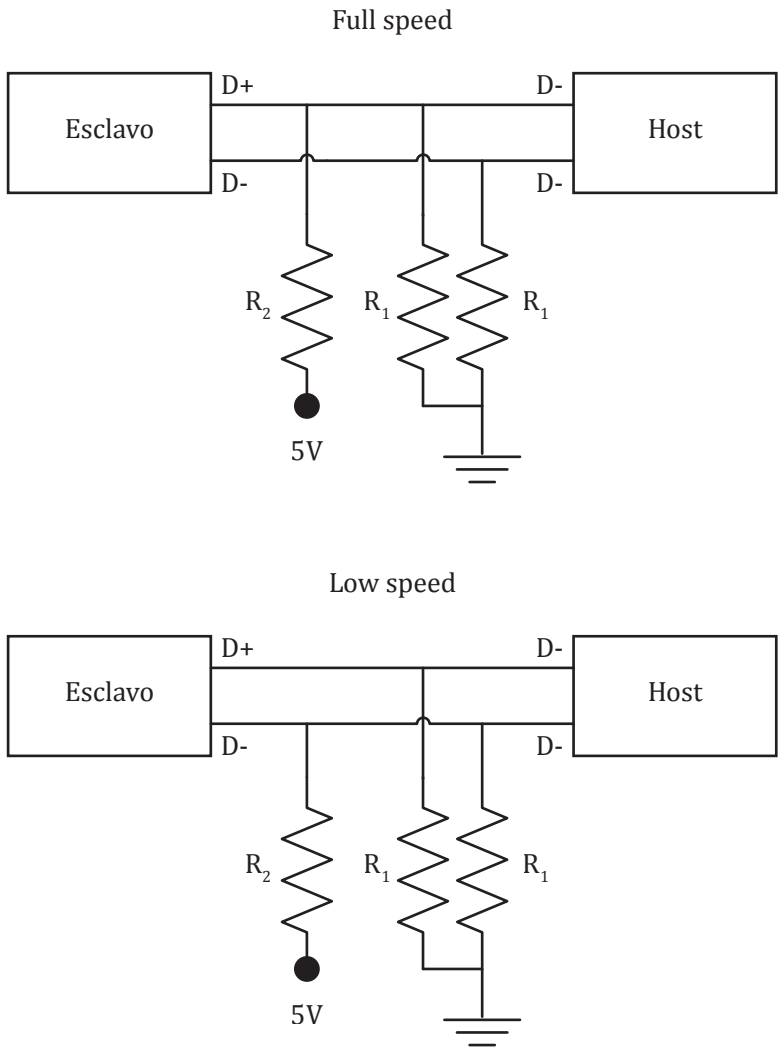
Elaboración propia.

Las longitudes de los cables que conectan a los dispositivos periféricos y el *Host* dependen del tipo de velocidad utilizada. Para el caso específico de la baja velocidad (1.5 Mbps), la longitud máxima permitida es de 3 metros, mientras que en velocidad media (12 Mbps) la longitud puede llegar a ser de hasta 5 metros. En la práctica es posible conectar hasta 5 *Hubs* en cascada para lograr tener una longitud mayor entre el dispositivo periférico y el *Host*. Esto permitiría obtener una distancia máxima de 30 metros.

61 Cfr. Axelson 2009.

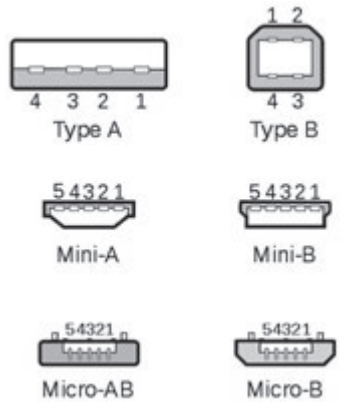
El tipo de conector físico que presentan tanto el *Host* como el dispositivo esclavo difiere en forma y en tamaño. El *Host* utiliza un conector USB tipo A, mientras que el esclavo utiliza el de tipo B. Dentro de cada tipo existen formatos estándar como los mini A o mini B y micro A o micro B usados en diferentes clases de dispositivos. Por ejemplo, una computadora de escritorio posee un conector estándar USB tipo A, mientras que los conectores mini A y mini B se encuentran principalmente en dispositivos OTG, como son los PDA o *tablets*. En el caso del dispositivo esclavo, el tamaño del conector USB tipo B puede variar de acuerdo con el tipo de aplicación. Por ejemplo, las impresoras y escáneres utilizan el conector estándar USB B, mientras que algunos dispositivos periféricos, como el Pickit 2 o el Pickit 3, emplean el conector mini USB. Otros dispositivos más compactos, como los teléfonos celulares, usan un conector micro USB para comunicarse con el computador. En el gráfico 14.3 se muestra el esquema de los tres tipos de conectores y su diagrama de pines.

Gráfico 14.2. Modos de configuración de las resistencias de *Pull Up* (R_2) para los modos *Full Speed* y *Low Speed*



Elaboración propia.

Gráfico 14.3. Los conectores USB tanto para el *Host* (tipo A) como para el dispositivo esclavo (tipo B). Los conectores mini y micro son de 5 contactos



Elaboración propia.

Las clases de conectores mini A y B y micro A y B contienen un quinto pin que cumple con la función de apantallamiento. Esto corresponde a una malla conductora que rodea toda la longitud de los cables D+ y D- con el fin de blindarlos contra la interferencia electromagnética del medio. Este blindaje es muy utilizado para las velocidades medias y altas, con la desventaja de que la flexibilidad del cable se ve un poco reducida, comparada con la de un cable no blindado[62].

La numeración de los pines sigue el orden que se observa en el gráfico 14.4. El conector provee las señales de 5 V y GND que pueden ser utilizadas por el dispositivo esclavo como fuente de energía. Por ejemplo, el Pickit 2 utiliza esta fuente no solo para alimentar su circuito interno, sino también para energizar el módulo de desarrollo donde se encuentra alojado el sistema embebido a programar. Este módulo puede contener pantallas LCD, Displays 7 segmentos, un circuito integrado MAX232, LEDs, entre otros, y todos estos elementos pueden ser activados por la fuente de energía del bus USB, siempre y cuando la carga total no exija más de 500 mA.

Gráfico 14.4. Diagrama de pines de los conectores USB

PIN	Función
1	5 V
2	D+
3	D-
4	GND
5	Apantallamiento

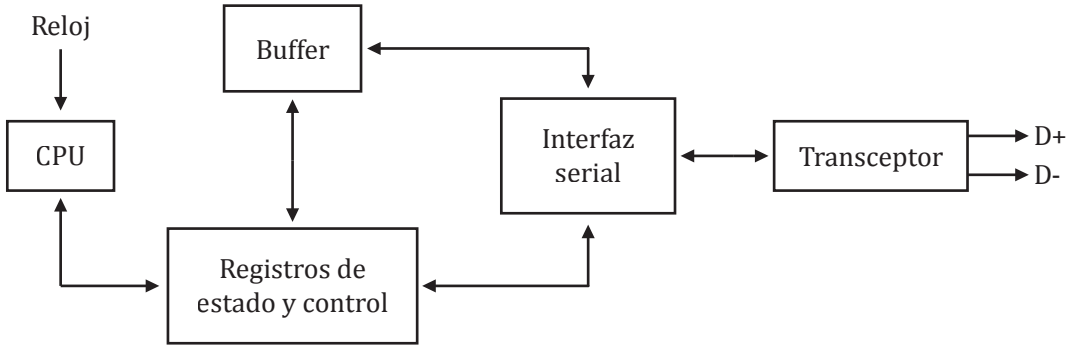
Elaboración propia.

Las líneas 3 y 4 de los conectores del gráfico 14.3 corresponden a D+ y D-, respectivamente. Finalmente, el quinto pin corresponde al apantallamiento que, según el diseño del dispositivo, puede ir conectado a un circuito especial o simplemente a la carcasa del dispositivo.

62 Cfr. Axelson 2009.

Por otra parte, de lado del dispositivo esclavo USB, la constitución del *hardware* debe poseer los siguientes elementos: un transceptor, un motor serial de interfaz USB, un *Buffer*, registros de estado y control y un procesador capaz de manejar la comunicación y funcionalidad del sistema. Por ejemplo, el PIC18F4550 contiene todos los elementos que se observan en el gráfico 14.5.

Gráfico 14.5. Elementos constitutivos del *hardware* de un dispositivo periférico USB



Fuente: Hoja de datos del PIC18F4550.

El transceptor USB es el circuito de interfaz que permite transformar las tensiones de las líneas en modo diferencial (D+ y D-) a un modo unipolar y viceversa. Este circuito, además, contiene las resistencias de *Pull up*, que se pueden habilitar de manera configurable de acuerdo con la velocidad de operación deseada. El transceptor es un dispositivo simple, que solo se encarga de adaptar los niveles de voltaje.

Por otra parte, la interfaz serial es el bloque que se conecta al transceptor para recibir y generar las señales a ser enviadas en el bus USB. Este bloque cumple con diversas funciones, entre las que se encuentran las siguientes:

1. Detectar los paquetes de entrada enviados por el *Host*.
2. Enviar paquetes al *Host*.
3. Detectar y enviar paquetes de inicio, fin, Reset, CRC, ACK, datos y demás en el bus.
4. Generar y detectar las tramas de *Bit Stuffing*[63].
5. Verificar y generar los valores CRC (bits de detección de error).
6. Verificar y generar los paquetes de identificación.
7. Interactuar con los registros de estado y control y almacenar la información recibida en el *Buffer*, al mismo tiempo que utiliza la información del *Buffer* para la construcción de paquetes para su transmisión.

El *Buffer* es otra parte muy importante, que permite alojar la información en un espacio de memoria SRAM que la interfaz serial transmitirá al bus USB. De la misma manera, la información recibida quedará almacenada en el *Buffer*. En el protocolo USB, el dispositivo esclavo debe contener una serie de datos que deben estar almacenados en el *Buffer*. Este conjunto de información, que corresponde a los descriptores, requiere de una cierta cantidad de memoria disponible.

63 Es un bit de nivel '0' que ingresa a la trama de la señal cuando hay demasiadas secuencias de '1' lógicos de manera consecutiva.

Los registros de estado y control permiten informar al CPU sobre la condición del bus USB y realizar, a su vez, cierto control, como la transferencia de paquetes (cuando el *Host* lo requiera) o la configuración de cierto estado del dispositivo esclavo.

Finalmente, el CPU es el procesador encargado de dar funcionalidad a la información recibida y transmitida en el bus USB a través de la interfaz serial. Es muy importante que el CPU pueda trabajar a una frecuencia de reloj proporcional a la requerida por la interfaz serial para producir tasas de 1.5 Mbps, 12 Mbps o 480 Mbps (aunque el PIC18F4550 solo soporta las primeras dos velocidades).

14.2 Modos de transferencia

En el protocolo USB, un paquete básico de comunicación se denomina «transferencia». Dentro de este paquete se encuentran muchos otros subpaquetes con información relevante. Una transferencia es un formato definido de datos que se compone de bits de dirección y verificación de error e información de control y estado. No todas las transferencias poseen los mismos campos, estos varían con el tipo de transferencia y la dirección de la información.

El *Host* siempre inicia las transferencias, ya sean estas de lectura o de escritura. Un dispositivo esclavo solo puede generar una transferencia si es que previamente ha recibido una solicitud por parte del maestro. Para tal fin, el dispositivo esclavo cuenta con uno o varios *Buffers* especiales denominados *Endpoints*. Todo *Endpoint* se caracteriza por la cantidad de bytes que puede almacenar y por un número y dirección que lo identifican en el bus. En una transferencia generada por el *Host*, la información recibida por el dispositivo esclavo queda almacenada en su *Endpoint*. De la misma manera, cuando el dispositivo esclavo tiene datos para transmitir, previamente almacena la información en el *Endpoint* para que sea transmitida cuando el *Host* lo decida. Todos los dispositivos esclavos poseen, al menos, un *Endpoint*: el *Endpoint* 0, que es usado para la configuración y extracción de información del dispositivo. Normalmente, los dispositivos esclavos cuentan con un *Endpoint* adicional. Algunos periféricos contienen dos *Endpoints* adicionales. Esto ocurre en el caso de dispositivos compuestos, como las impresoras que incorporan la función de *Scanner*.

Las transferencias tienen como origen y destino un *Endpoint*. De esta manera, existen transferencias de maestro a esclavo (denominadas «transferencias tipo OUT») y de esclavo a maestro (denominadas «transferencias tipo IN»). Solo los dispositivos esclavos cuentan con un *Endpoint*. El *Host* no presenta *Endpoints*[64].

Según la función y prioridad, existen cuatro tipos de transferencias en el bus USB:

1. Control: que es la utilizada por todos los tipos de dispositivos USB y sirve para que el *Host* pueda solicitar los descriptores al esclavo.
2. Bulk: es una transferencia que permite el envío de la información de maestro a esclavo y viceversa de forma rápida, pero con ciertos retrasos en algunos paquetes si el tráfico en el bus es elevado.
3. Interrupción: este tipo de transferencia garantiza un tiempo de llegada de la información, pero es utilizada en dispositivos lentos como ratones o teclados.
4. Isócrona: este tipo de transferencia sí garantiza el tiempo de llegada de la información y una alta tasa de transferencia. Sin embargo, no presenta tramas de verificación de error, de manera que un

64 Cfr. Axelson 2009.

bit errado tendrá que ser tomado como válido. Esta transferencia se utiliza, por ejemplo, en dispositivos de audio.

El tráfico del bus USB se maneja a través de unidades pequeñas de información denominadas «microtramas». Una microtrama tiene un tiempo de duración de 1 milisegundo para la baja velocidad (*Low Speed*) y de 125 microsegundos para la velocidad media (*Full Speed*). Cada microtrama comienza con un bloque SOF (Inicio de trama o *Start of a Frame*, del inglés). Una transferencia se puede dividir en múltiples microtramas de forma consecutiva. Ninguna otra transferencia se puede interrumpir hasta que se hayan completado todas las demás microtramas.

Las transferencias se dividen en bloques más pequeños denominados «transacciones». Una transacción contiene paquetes de información que van en dos sentidos (de maestro a esclavo y viceversa). Además, muestra el número de *Endpoint* destino y su dirección. El número de *Endpoint* es un valor entre 0 y 15. La dirección puede ser IN o OUT.

Las transacciones ocurren en fases. Estas pueden ser Paquete TOKEN, Paquete de datos o Paquete Handshake. Asimismo, cada paquete contiene bloques de códigos binarios que especifican información. En el gráfico 14.6 se pueden observar todos los subbloques de una transferencia.

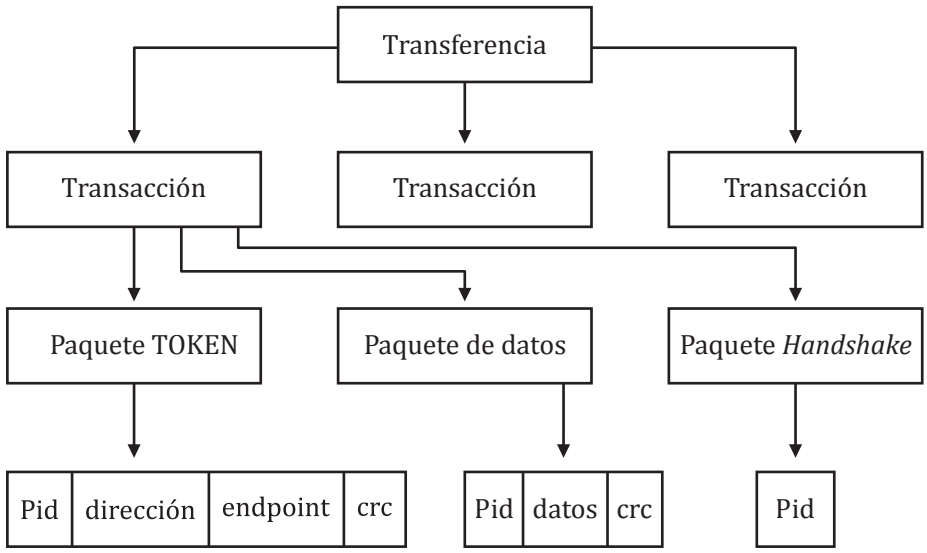
Los paquetes TOKEN tienen como objetivo delimitar el inicio de una transferencia. Se dividen en cuatro campos: PID, Dirección, *Endpoint* y CRC. El campo PID siempre es generado por el *Host* y puede corresponder a cuatro tipos: OUT, IN, SOF o SETUP. Los dos primeros especifican cuál de los elementos de la comunicación será el transmisor, el tipo SOF implica el inicio de una nueva transacción y el bloque SETUP implica que la transacción tiene fines de configuración del *Endpoint* y su dirección de acceso.

El campo Dirección especifica el dispositivo a acceder en el bus USB y el campo *Endpoint* contiene el número de *Endpoint* del dispositivo. Finalmente, la trama CRC contiene bits que permiten verificar si se han recibido correctamente los datos sin ningún bit errado. En caso de que se detecte un error, se puede solicitar el reenvío de la información desde el dispositivo esclavo.

El Paquete de datos contiene la información a enviar del *Host* al dispositivo esclavo o desde el dispositivo hacia el *Host*. También puede incluir información de estado del dispositivo esclavo. Este Paquete de datos puede ser de cuatro tipos: DATA0, DATA1, DATA2 o MDATA. En este caso, el campo PID especifica uno de estos cuatro tipos con solo cuatro bits. Los primeros dos tipos especifican un tipo de paquete que contiene bits de redundancia para evitar pérdidas de datos; y el tipo DATA2, transferencias isócronas y el tipo MDATA transferencias isócronas divididas en múltiples transacciones.

El Paquete *Handshake*, por otra parte, permite determinar el resultado de la recepción de los paquetes anteriores. Este paquete solo contiene un campo PID, que corresponde al código del estado de la transacción.

Gráfico 14.6. Estructura de una transferencia de datos en el bus USB



Elaboración propia.

Entre los tipos de Paquete *Handshake* se tienen los siguientes:

1. ACK: confirma que el paquete ha sido recibido adecuadamente.
2. NAK: indica que el receptor no ha podido aceptar los datos.
3. STALL: indica que el requerimiento de control no se encuentra soportado por el dispositivo.
4. NYET: el dispositivo acepta un paquete sin errores, pero no se encuentra disponible para aceptar un nuevo paquete aún.
5. PRE: es un bloque de bits de preámbulo que indican que el siguiente paquete será enviado a baja velocidad.
6. ERR: este bloque solo es generado por un *Hub* para indicar que se ha detectado un error en una transacción.
7. PING: es una bandera de ocupado para las transferencias de control y *Bulk* generado después del NYET.

Antes de que se produzca una transferencia, el *Host* y el dispositivo esclavo deben establecer una tubería (del inglés *Pipe*). Esto significa que debe crearse una asociación entre el *Endpoint* del dispositivo esclavo y el *software* controlador del *Host* USB. Esta asociación se produce en el proceso de enumeración que se detalla en el siguiente subcapítulo. Todos los dispositivos esclavos presentan una tubería de control por defecto para el *Endpoint* número 0.

14.3 El proceso de enumeración

Antes de que el *Host* se pueda comunicar con un dispositivo esclavo, debe primero enumerar a dicho dispositivo, lo cual implica aprender sus características. Este proceso involucra por parte del *Host* la asignación de una dirección al esclavo, leer sus descriptores, asignar y cargar un *driver* controlador en el sistema operativo del computador y seleccionar y configurar los requerimientos de energía del dispositivo esclavo.

El *Host* detecta si hay nuevos dispositivos conectados al bus mediante transferencias continuas de control al *Endpoint* 0. Desde la perspectiva del usuario, la enumeración culmina con el reconocimiento del dispositivo en la herramienta «Administrador de dispositivos» del sistema operativo Windows. Solo la primera vez que el dispositivo periférico se conecta al computador, el usuario deberá ubicar la ruta del disco duro donde se encuentra el *driver*.

El proceso de enumeración se divide en doce pasos, que se describen a continuación en el orden en que se ejecutan[65]:

1. El usuario conecta un nuevo dispositivo al bus USB, el cual entrega hasta 100 mA de corriente al dispositivo esclavo para su funcionamiento.
2. El *Hub* raíz detecta la presencia del dispositivo esclavo conectado monitoreando el voltaje en las líneas D+ y D-. El *Hub* raíz contiene resistencias de *Pull down* de valor entre 14.25 kΩ y 24.8 kΩ conectadas a las líneas D+ y D-. El dispositivo esclavo tendrá una resistencia de *Pull up* con un valor entre 900 Ω y 1.5 kΩ en la línea D+ (*Full Speed*) o en la línea D- (*Low Speed*). Al conectar estas resistencias de *Pull up* al puerto, la línea respectiva se llevará a un nivel '1', lo cual permite detectar al dispositivo y conocer su tipo de velocidad.
3. El *Host* comienza a aprender del nuevo dispositivo mandándole un comando *Get Port Status* a su *Hub* raíz. Este retorna el momento en el cual ha detectado el nuevo dispositivo.
4. El *Hub* raíz detecta si el dispositivo es de baja o media velocidad examinando los niveles de voltaje con referencia a GND en las líneas D+ y D-.
5. El *Hub* raíz reinicia el dispositivo esclavo. La condición de reinicio consiste en fijar las líneas D+ y D- en nivel '0' por al menos 10 milisegundos. Este proceso solo se realiza al dispositivo recientemente conectado. Los demás dispositivos conectados al bus no se ven afectados.
6. En caso de que el dispositivo esclavo no sea de baja velocidad, el *Host* deberá averiguar si puede soportar la alta velocidad (*High Speed*). Para esto, el *Hub* raíz enviará secuencias denominadas Chirp J (variación del estado de la línea D+) y Chirp K (variación del estado de la línea D+) a dos velocidades diferentes. Si el *Host* recibe respuesta a la secuencia de alta velocidad, el dispositivo será reconocido como *Full Speed*.
7. El *Hub* raíz establece un camino para la señal del dispositivo en el bus. El *Host* finaliza el estado de Reset y verifica la salida de este estado enviando un comando *Get Port Status* al dispositivo esclavo. Una vez que el periférico responde con un bit, ingresa a un estado denominado *Default*, en el cual se encuentra capacitado para responder a transferencias de control dirigidas al *Endpoint* 0. De esta manera, el *Host* se comunica con el periférico a través de una dirección por defecto de valor 0x00.

65 Cfr. Axelson 2009.

8. El *Host* transmite el comando *Get Descriptor*, con el fin de conocer las características del dispositivo esclavo y el tamaño máximo de los paquetes que serán enviados por el periférico.
9. El *Host* asigna una nueva dirección al dispositivo esclavo. Esta dirección se mantendrá firme hasta que el usuario reconecte el dispositivo al bus o se reinicie el computador. En este momento el esclavo entra en un estado denominado *Address*.
10. El *Host* solicita el descriptor mediante el comando *Get Descriptor Request* a la nueva dirección. Esto contiene información básica del dispositivo. Por lo general, existe más de un solo descriptor. Por lo tanto, todos estos serán enviados al *Host*.
11. El *Host* asigna un *driver* de dispositivo. Después de que el *Host* ha analizado los descriptores del dispositivo esclavo, debe buscar un *driver* que se pueda hacer cargo de la comunicación con el periférico. En el sistema operativo Windows se utilizan archivos de extensión .INF (información) que ayudan a encontrar el *driver* en el disco duro. Este archivo debe ser ubicado en la ruta del disco duro por el usuario solo la primera vez que el dispositivo ha sido conectado al bus.
12. El *driver* seleccionado busca la mejor configuración para el dispositivo y la mantiene registrada en el sistema operativo. Cuando el dispositivo se vuelva a conectar en un futuro, esta configuración será la utilizada por defecto.

14.4 Las clases de descriptores

Los descriptores son arreglos de datos que contienen campos específicos con información del dispositivo esclavo. Esta información reside en una memoria no volátil para luego alojarse en el *Endpoint* 0 del dispositivo. Cada descriptor maneja un tipo específico de información sobre el dispositivo. Existen en total un máximo de 18 tipos de descriptores. No todos los descriptores son necesarios, ya que presentan información que no necesariamente corresponde al dispositivo. Los descriptores más utilizados se observan en el gráfico 14.7.

Gráfico 14.7. Tipos de descriptores

Descriptor	Uso obligatorio	Descripción
Dispositivo	SÍ	Es el primer descriptor a ser leído. Contiene información general del dispositivo.
Configuración	SÍ	Contiene información del número de interfaces soportadas y los requerimientos de energía.
Cadena de caracteres	NO	Contiene texto descriptivo en formato ASCII que permite mostrar información del dispositivo al usuario.
Interface	SÍ	Contiene información sobre la función del dispositivo.
Endpoint	SÍ	Contiene información de todos los Endpoints adicionales al 0.

Descriptor	Uso obligatorio	Descripción
Calificador de dispositivo	SÍ	Contiene información de protocolos, clases y tamaño del *Endpoint* 0.
Otras configuraciones de velocidad	SÍ	Muy similar al descriptor de configuración. Contiene información de interfaces soportadas a otras velocidades.
OTG	NO	Usados por dispositivos que poseen la funcionalidad adicional OTG.

Elaboración propia.

El descriptor de dispositivo es el primer bloque a ser enviado al *Host* para la identificación del dispositivo por el sistema operativo. En el gráfico 14.8 se muestran los campos de este descriptor.

Gráfico 14.8. Campos del descriptor de dispositivo

Campo	Tamaño en bytes	Descripción
bLength	1	Tamaño del descriptor en bytes (0x12).
bDescriptorType	1	La constante de dispositivo (0x01).
bcdUSB	2	Número de especificación USB (BCD).
bDeviceClass	1	Código de clase.
bDeviceSubclass	1	Código de subclase.
bDeviceProtocol	1	Código del protocolo.
bMaxPacketSize0	1	Máximo tamaño de un paquete para el *Endpoint* 0.
idVendor	2	ID del vendedor.
idProduct	2	ID del producto.
bcdDevice	2	Número de dispositivo (BCD).
iManufacturer	1	Índice al descriptor de cadena de caracteres con la información del fabricante.
iProduct	1	Índice al descriptor de cadena de caracteres con la información del producto.
iSerialNumber	1	Índice al descriptor de cadena de caracteres con la información del número serial.
bNumConfigurations	1	Número de posibles configuraciones.

Elaboración propia.

Como se puede observar, cada campo es fijo y contiene un tamaño en bytes específico. Cada uno de los campos tiene valores determinados para el dispositivo, mientras que otros pueden ser modificados por el diseñador del dispositivo USB. Los dos primeros campos siempre mantienen el mismo valor. El campo *bcdUSB* especifica el tipo de versión en código BCD del formato USB. Por ejemplo, un código 0x06 implica el estándar USB 1.1, mientras que el estándar USB 2.0 se especifica con el código 0x2000.

El campo *bDeviceClass* detalla la clase de dispositivo al cual corresponde el periférico. Los dispositivos de clase especial o definida por el usuario deben contener el código 0xFF en este campo. Entre las clases se encuentran los tipos generales de dispositivo, los cuales corresponden a equipos de interfaz de usuario, equipos de comunicaciones, *Hub*, dispositivo de cuidado de salud, dispositivo de diagnóstico, dispositivo inalámbrico, misceláneo o definido por el usuario. De la misma manera, el campo *bDeviceSubclass* permite especificar la subclase del dispositivo dentro de su clase[66].

El campo *bDeviceProtocol* especifica un protocolo para la clase y subclase seleccionada. Por otro lado, el campo *bMaxPacketSize0* detalla el máximo tamaño de paquete soportado por el *Endpoint* 0.

El campo *idVendor* es asignado por la USB-IF[67] cuando un fabricante va a comercializar su producto. El código asignado por la corporación requiere del pago de una anualidad por parte del fabricante. El campo *idProduct* puede ser asignado por el propio fabricante para identificar cada uno de sus modelos de producto. De la misma manera, el campo *bcdDevice* es asignado por el fabricante para ayudar al *Host* a seleccionar un *driver*.

Cada descriptor, entonces, debe poseer campos fijos y otros que pueden ser modificados por el programador. La estructura del descriptor no puede cambiar, ya que esto afectaría directamente el proceso de enumeración.

14.5 El módulo USB del PIC18F4550

El PIC18F4550 contiene un motor de interfaz serial SIE (de las siglas en inglés *Serial Interface Engine*), el cual es compatible con el estándar USB 2.0 en las velocidades baja y media. Esto permite que el PIC pueda comunicarse con un computador a través del puerto USB.

El SIE se conecta directamente al puerto USB a través de un transceptor interno que maneja las líneas D- (RC4) y D+ (RC5) del PIC18F4550 (aunque también es posible conectar un transceptor externo a los pines del SSP). Adicionalmente, se cuenta con un regulador interno de 3.3 V, cuya función es alimentar el transceptor interno y a su vez servir como tensión para las resistencias internas de *Pull up* que configuran la velocidad de comunicación. La tensión de salida del regulador se conecta al pin V_{USB} del PIC18F4550, donde se recomienda la conexión de un condensador de desacoplo de 220nF ±20%. Las resistencias de *Pull up* conectadas a los pines D+ y D- pueden ser tanto internas como externas. En caso de que estas resistencias sean conectadas externamente, deberán acoplarse al pin V_{USB}, ya que presenta una tensión de 3.3 V. Se debe tomar en cuenta que la habilitación del regulador interno depende de la configuración inicial del código del programa en MPLAB X cuando se utiliza la directiva *#pragma config VREGEN = ON*. La especificación de los niveles de tensión USB requiere del uso de 3.3 V en las resistencias de *Pull up*. Sin embargo, el resto de la circuitería del PIC puede operar a 5 V.

La memoria SRAM del PIC18F4550 presenta 1 kB dedicado especialmente para el uso directo del SIE. Esta memoria se utiliza para almacenar la información de los descriptores y los *Endpoints*. Son precisamente los bancos del 4 al 7 los que se encuentran conectados a un puerto dual que permite que este espacio de memoria pueda servir como propósito general o como función específica del SIE. En el gráfico 14.9 se muestra un diagrama de bloques del módulo USB interno del PIC18F4550. El puerto SSP (de las siglas en inglés *Streaming Parallel Port*) permite la conexión de un transceptor USB externo y soportar grandes volúmenes de datos a través de este.

66 Cfr. Axelson 2009.

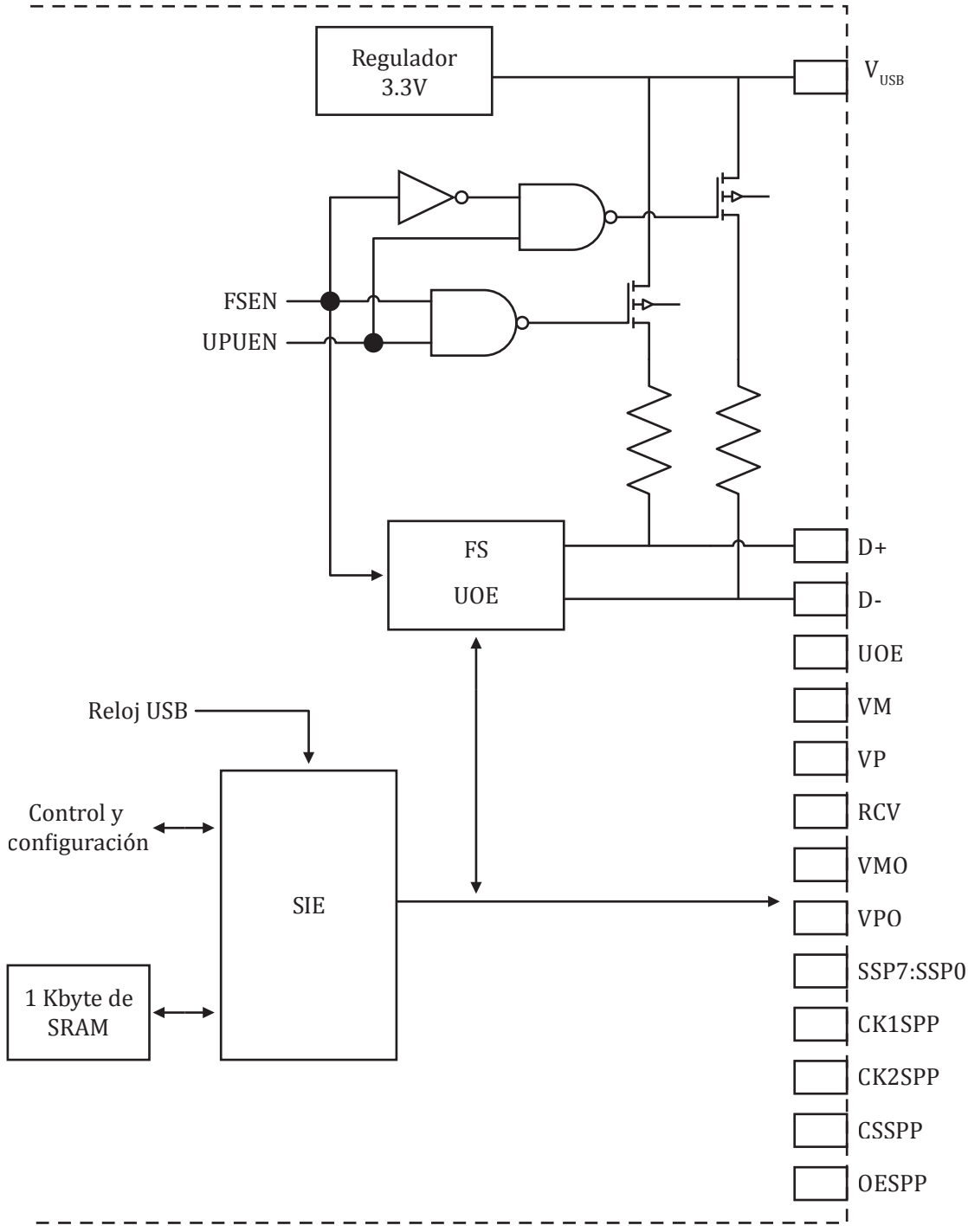
67 Corporación internacional encargada del cumplimiento de estándares de los dispositivos USB.

El controlador USB del PIC18F4550 soporta los 4 tipos de transacción (Control, Isócrona, *Bulk* y de Interrupción) y hasta 30 direcciones de *Endpoint*, además del *Endpoint* 0. Claro está que existe una dirección para una transacción del tipo IN y otra para la transacción de tipo OUT sobre el mismo número de *Endpoint*. Si se requiere configurar el módulo USB para una transferencia Isócrona, es necesario el uso exclusivo del módulo SSP. Básicamente, cada *Endpoint* es un *Buffer* que almacena información dentro del PIC. Cada *Endpoint* posee un identificador y un registro de configuración independiente. Además, se puede configurar como entrada (IN) o salida (OUT).

Por cada *Endpoint* utilizado se deberá reservar una cierta cantidad de memoria SRAM para el almacenamiento del descriptor y para un *Buffer*. Toda la operación del módulo USB se obtiene mediante tres registros de control. En total existen 22 registros para configurar completamente el módulo USB del PCI18F4550:

1. Registro de control UCON
2. Registro de configuración UCFG
3. Registro de estado de transferencia (USTAT)
4. Registro de dirección de dispositivo (UADDR)
5. Registros de número de trama (UFRMH:UFRML)
6. Registros habilitadores de *Endpoint* 0 al 15 (UEPn, donde n puede ser desde 0 hasta 15)

Gráfico 14.9. Diagrama de bloques del módulo USB del PIC18F4550



Elaboración propia.

El bit USBEN del registro UCON es el más prioritario al comienzo de la configuración del módulo USB, ya que permite habilitar o deshabilitar el módulo cuando se halla en nivel '0'. En el registro UCFG se encuentran el control de la velocidad media o baja y de la configuración de las resistencias de *Pull up*. Con el bit FSEN de este registro se logra configurar el modo de velocidad. Con un nivel '1' lógico se selecciona el modo de *Full Speed*, lo cual requiere una frecuencia de reloj configurada a 48 MHz, mientras que con un nivel '0' lógico se selecciona el modo *Low Speed*, el cual requiere una frecuencia de reloj del USB de 6 MHz. Por otra parte, el bit UPUEN permite habilitar las resistencias de *Pull up* internas del módulo USB con un nivel '1'.

El PIC18F4550 cuenta con múltiples fuentes de interrupción y una única prioridad de interrupción, que puede ser baja o alta para todos los eventos USB. El registro PIR2 contiene el bit de bandera USBIF, encargado de indicar cualquier evento relacionado con el bus USB. El registro PIE2 incluye el bit USBIE, que permite habilitar las interrupciones del bus; USB mientras que el registro IPR2 contiene el bit USBIP, que asocia las interrupciones a baja o alta prioridad.

14.6 La librería MLA (*Microchip Library Application*) para dispositivos USB de la marca Microchip

El manejo de la interfaz USB del PIC18F4550 es algo tedioso, sobre todo si se intenta diseñar el *firmware* siguiendo la metodología vista en los programas de los capítulos anteriores. La cantidad de configuraciones y códigos requeridos para una aplicación USB requiere de mucho tiempo y conocimiento detallado del funcionamiento del protocolo. Esto, además, obligaría al programador a utilizar horas de desarrollo de código y verificaciones para comprobar que su *firmware* se encuentra operando de manera adecuada. Para evitar todos estos inconvenientes y acelerar la generación de aplicaciones USB, Microchip ha elaborado una librería gratuita para las familias PIC18F, PIC24, dsPIC30 y PIC32, que permite hacer uso de funciones y macros que facilitan el manejo del puerto USB y otros periféricos como memorias SD, pantallas LCD gráficas y táctiles, comunicación a través de una red LAN Ethernet y Wifi.

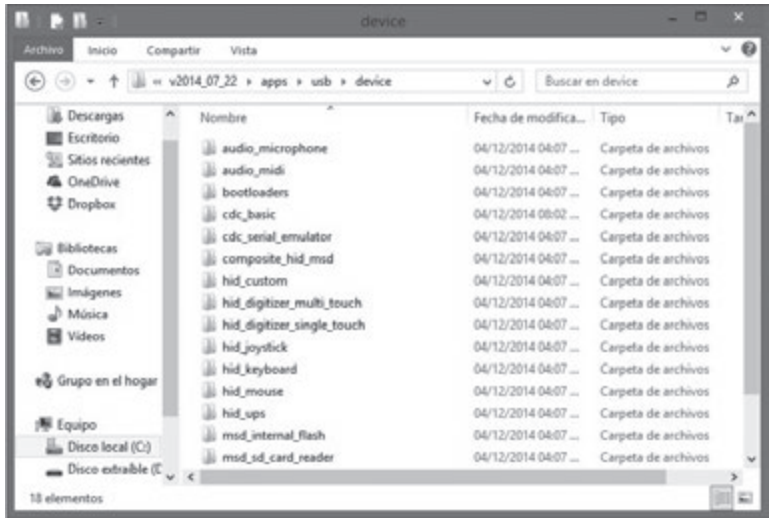
La librería MLA se descarga gratuitamente de la página web de Microchip . La descarga consiste en un archivo de extensión .EXE que, al ser ejecutado, instala una serie de proyectos compuestos por ejemplos para diferentes aplicaciones y modelos de microcontroladores en el directorio raíz del disco duro. Hasta la fecha, la última versión se encuentra actualizada al 22 de julio de 2014. Dentro de la ruta se encontrará una serie de ejemplos para ser usados con MPLAB X y el compilador XC8. Estos se hallan separados por clases de dispositivo. Por ejemplo, existen programas para el manejo de la clase HID (simulación de un teclado, ratón y *joystick* virtual), programas para el manejo de la clase CDC (conversión USB-RS232) y programas para el manejo de la clase MSD de almacenamiento masivo. En el gráfico 14.10 se muestra la estructura del directorio raíz con los diversos proyectos de ejemplo que Microchip brinda. Cada ejemplo contiene un proyecto que puede ser adaptado a diferentes modelos de microcontroladores definidos para un módulo de desarrollo específico que Microchip ofrece.

Una vez instalada la librería MLA, se deberá abrir un ejemplo específico. Por ejemplo, la carpeta *cdc_basic* contiene proyectos de ejemplo para emular un puerto RS232 en diversos microcontroladores de Microchip. En el caso específico del PIC18F4550, Microchip ofrece un módulo de desarrollo deno-

minado el PICDEM FS USB basado en el PIC18F4550 en empaque de montaje superficial de 44 pines. La tarjeta incluye pulsadores, LED, un conector USB, un puerto RS232, un cristal de 20 MHz, un conector de depurador ICD2, un sensor de temperatura, un potenciómetro y otros elementos más. Cuenta con lo básico para implementar un sistema de comunicación USB. Los programas de ejemplo para el PIC18F4550 son basados en este módulo de desarrollo, por lo cual se recomienda descargar de internet el manual de usuario de la tarjeta para realizar las modificaciones del caso y adaptar los ejemplos a un *hardware* específico (ya que los ejemplos utilizan funciones para el acceso a los pines de E/S que se encuentran predefinidos en función a las conexiones del PIC18F4550 en la placa PICDEM FS USB). En el gráfico 14.11 se muestra una imagen del módulo PICDEM FS.

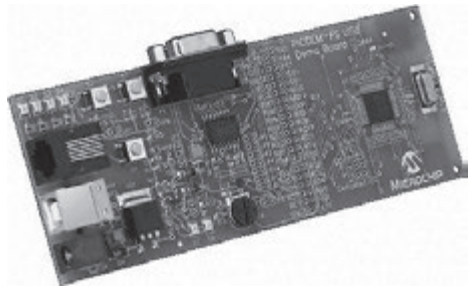
Para tener acceso a uno de los ejemplos, se deberá abrir el proyecto desde MPLAB X buscando la carpeta específica cdc_basic y en el combo box de configuración en la ventana de herramientas seleccionar PICDEM FSUSB. Tomar en cuenta que no todos los ejemplos pueden ser implementados en este módulo de desarrollo. Sobre todo los proyectos USB OTG, que son específicos de los PIC32. Una vez abierto un ejemplo, se tendrá la lista de librerías agregadas al proyecto en el inspector de archivos. El programa principal tendrá de nombre main.c y es aquí donde el programador podrá editar el código y crear las modificaciones necesarias para poder adaptar un ejemplo a las especificaciones del proyecto deseado. En el gráfico 14.12 se ofrece una imagen del proyecto cdc_basic, que muestra cómo conectar el PIC18F4550 al computador a través del puerto USB mediante la clase CDC, que permite que el computador reconozca al PIC como un puerto RS232.

Gráfico 14.10. Estructura de proyectos de ejemplo almacenados en el directorio raíz del disco duro



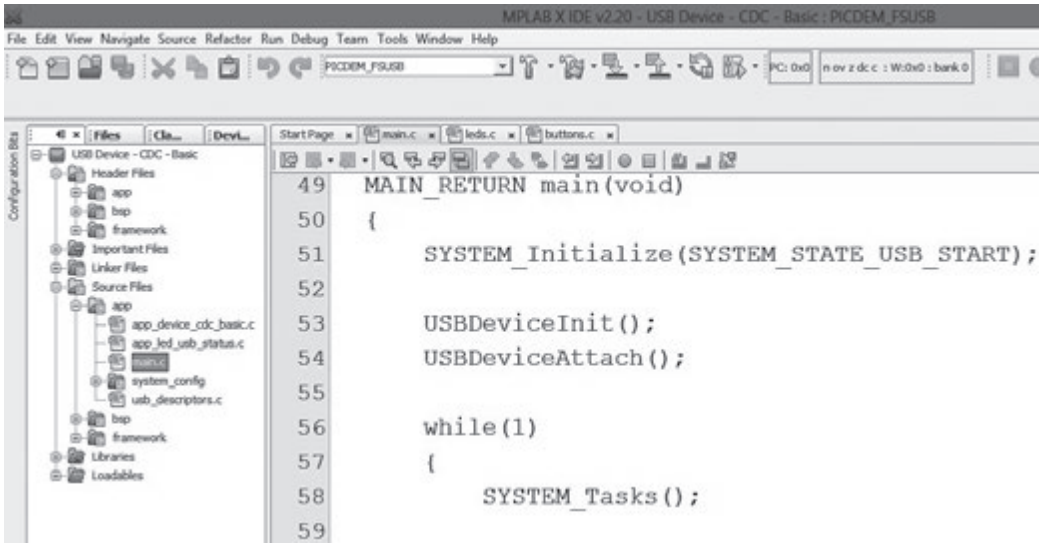
Elaboración propia.

Gráfico 14.11. Módulo de desarrollo PICDEM FS USB basado en el PIC18F4550



Fuente: http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=DM163025

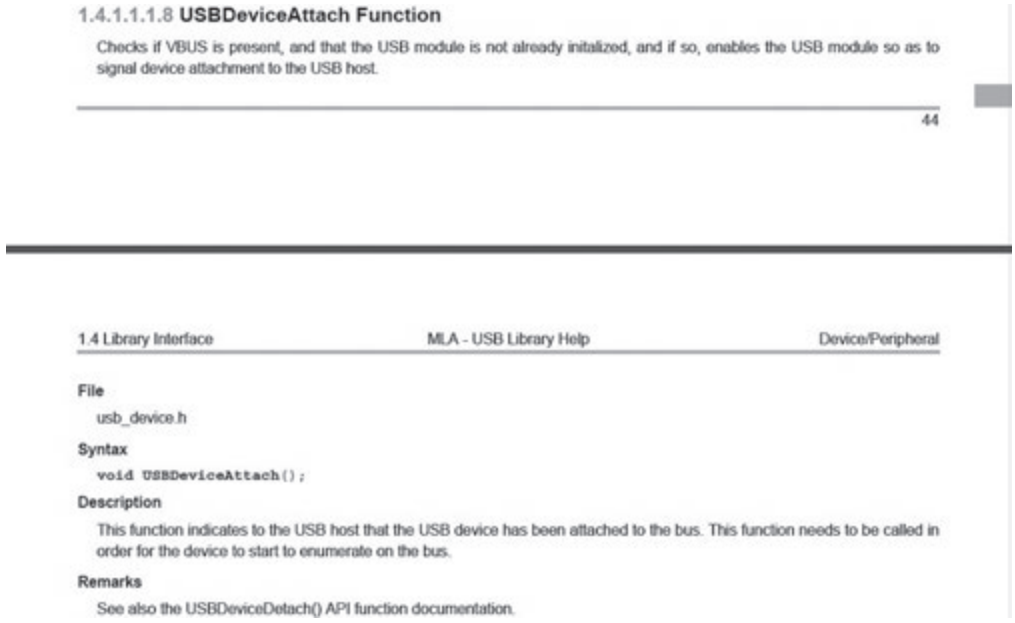
Gráfico 14.12. Proyecto de ejemplo cdc_basic



Elaboración propia.

Dentro del directorio del MLA en la ruta a «Microchip/mla/v2014_07_22/doc» se encuentran varios archivos en formato .pdf con información sobre las funciones y macros que utilizan los programas de ejemplo. El archivo help_mla_usb.pdf contiene una lista detallada de funciones, macros, módulos de desarrollo y guías de configuración de los proyectos de ejemplo. Con este documento se puede tener una referencia para entender cada programa de ejemplo paso a paso e identificar el funcionamiento de cada función dado en este paquete de proyectos demostrativos. En el gráfico 14.13 se muestra una imagen de este utilitario.

Gráfico 14.13. Utilitario de ayuda para obtener información de las funciones y macros de la librería MCHPFSUSB



1.4.1.1.1.8 **USBDeviceAttach Function**

Checks if VBUS is present, and that the USB module is not already initalized, and if so, enables the USB module so as to signal device attachment to the USB host.

44

1.4 Library Interface | MLA - USB Library Help | Device/Peripheral

File

usb_device.h

Syntax

```
void USBDeviceAttach();
```

Description

This function indicates to the USB host that the USB device has been attached to the bus. This function needs to be called in order for the device to start to enumerate on the bus.

Remarks

See also the USBDeviceDetach() API function documentation.

Referencia: help_mla_usb.pdf de Microchip.

14.7 Ejemplo de comunicación entre el PIC18F4550 y un computador a través del puerto USB

Partiendo del ejemplo que se encuentra en el conjunto de proyectos de Microchip, se toma como base el proyecto USB cdc_basic para presentar la comunicación serial virtual entre el PIC18F4550 y el computador. Al interconectar el PIC con el computador, este último realizará el proceso de enumeración, con lo cual solo la primera vez solicitará la ruta del archivo de extensión .inf que se encuentra dentro de una subcarpeta del proyecto «utilities/inf» con el nombre mchpcdc.inf. Posteriormente, el sistema operativo asignará un puerto COM virtual al PIC18F4550 y cada vez que este sistema embebido se conecte al puerto USB del mismo computador, dicho puerto virtual aparecerá activo.

Existen otras alternativas para resolver esta aplicación. La empresa FTDI provee el circuito integrado FT232RL, el cual permite interconectar cualquier microcontrolador al bus USB a través de un *driver* denominado VCP (del inglés *Virtual Com Port*). Este dispositivo opera en modo *High Speed*, con lo cual se pueden obtener tasas superiores a 921 600 bps. El FT232RL se conecta al microcontrolador a través de las líneas del EUSART (TX y RX), y como salida se conecta a las líneas D+ y D- del bus USB. Para el microcontrolador, la interfaz USB resulta transparente.

El ejemplo del proyecto cdc_basic realiza las siguientes operaciones:

1. Al conectar el PIC18F4550 a través del puerto USB al computador, este enumera el dispositivo y crea un puerto COM virtual.
2. El usuario deberá verificar el puerto COM asignado por el sistema operativo en el Administrador de dispositivos.
3. Abrir el Hyperterminal y el puerto COM asignado al PIC18F4550. La tasa de bit se deberá configurar a 9600 baudios, con un solo bit de parada y sin control de flujo.
4. Pulsar el pulsador conectado al pin RB4. Con esto aparecerá el mensaje "Hola Mundo\r\n" en el Hyperterminal.
5. Los datos enviados desde el Hyperterminal serán almacenados en un buffer de memoria SRAM del PIC18F4550; luego, cada byte será retransmitido al *Hos*t con un incremento de 1 dígito LSB.
6. Los LED conectados a los pines RB0 y RB1 estarán continuamente oscilando mientras no haya actividad en el puerto USB.

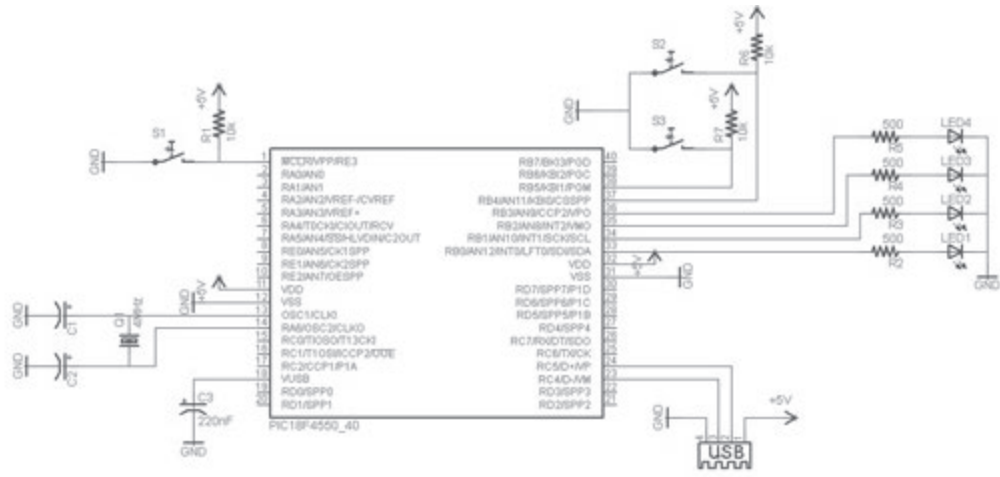
En el gráfico 14.14 se muestra el circuito esquemático necesario para probar el ejemplo sobre la clase CDC. Los pulsadores cuentan con una resistencia de *Pull up* externa, ya que es así como se encuentran configurados en el módulo PICDEM FS USB. Una de las principales modificaciones que se tienen que realizar en el programa es el divisor del PLL, el cual debe ser uno, ya que en todos los ejemplos del libro se ha trabajado con un cristal externo de 4 MHz en vez del de 20 MHz que contiene el módulo PICDEM. Esto se logra abriendo el archivo system.c que se encuentra en la ruta del inspector de archivos del MPLAB X mostrada en el gráfico 14.15. En el programa 14.1 se detalla la etapa de configuración del programa de ejemplo.

Programa 14.1. Etapa de configuración de los fusibles del programa de ejemplo para el uso de la clase CDC en el archivo system.c

```
#include <system.h>
#include <system_config.h>
#include <usb/usb.h>
/** CONFIGURATION Bits **********************************************/
#pragma config PLLDIV   = 1 // (cambiar de 5 a 1)
#pragma config CPUDIV   = OSC1_PLL2
#pragma config USBDIV   = 2
#pragma config FOSC     = HSPLL_HS
#pragma config FCMEN    = OFF
#pragma config IESO     = OFF
#pragma config PWRT     = OFF
#pragma config BOR      = ON
#pragma config BORV     = 3
#pragma config VREGEN   = ON
#pragma config WDT      = OFF
```

```
#pragma config WDTPS    = 32768
#pragma config MCLRE    = ON
#pragma config LPT1OSC  = OFF
#pragma config PBADEN   = OFF
#pragma config STVREN   = ON
#pragma config LVP      = OFF
#pragma config XINST    = OFF
#pragma config CP0      = OFF
#pragma config CP1      = OFF
#pragma config CPB      = OFF
```

Gráfico 14.14. Circuito esquemático para la prueba del ejemplo de la clase CDC



Elaboración propia.

Posteriormente, en el archivo main.c se encuentra la secuencia principal del programa que ejecutará el PIC18F4550. Esta sección del código se muestra en el programa 14.2. La función principal *main()* es muy sencilla, realiza la inicialización del módulo USB interno y de los pines periféricos a utilizar mediante la función *SYSTEM_Initialize()*, se encarga de configurar los pines del Puerto B como salidas (RB3 a RB0) y los pines RB4 y RB5 como entradas digitales. Luego, ejecuta una función *USBDeviceInit()* que inicializa el módulo USB para que todas sus variables internas estén en condición por defecto, y al mismo tiempo habilita la pila de memoria SRAM. Después, la función *USBDeviceAttach()* permite que el *Host* pueda realizar el proceso de enumeración del PIC18F4550 inicializando la interrupción del bus USB y configurando las resistencias de *Pull up* de los pines D+ y D-. La función *USBDeviceTasks()* opera cuando se define la constante USB_POLLING, la cual especifica que esta función deberá ejecutarse periódicamente para tomar control de las transferencias necesarias para llevar a cabo el proceso de comunicación una vez realizada la enumeración con el *Host*. En el programa 14.2 se muestra el código de la función principal.

Gráfico 14.15. Ruta de acceso para abrir el archivo system.c en el proyecto cdc_basic

Elaboración propia.

Programa 14.2. Función principal main()

```
#include <system.h>
#include <system_config.h>
#include <app_device_cdc_basic.h>
#include <app_led_usb_status.h>
#include <usb/usb.h>
#include <usb/usb_device.h>
#include <usb/usb_device_cdc.h>
MAIN_RETURN main(void)
{
  SYSTEM_Initialize(SYSTEM_STATE_USB_START);
  USBDeviceInit();
  USBDeviceAttach();
  while(1)
  {
    SYSTEM_Tasks();
    #if defined(USB_POLLING)
      USBDeviceTasks();
    #endif
    if( USBGetDeviceState() < CONFIGURED_STATE )
    {
      continue;
    }
```

```
    if( USBIsDeviceSuspended()== true )
    {
      continue;
    }
    //Application specific tasks
    APP_DeviceCDCBasicDemoTasks();
  }//end while
}//end main
```

Posteriormente, la función *USBGetDeviceSta*te() retorna el estado para que el módulo USB del PIC18F4550 se pueda comunicar en el bus. Si la función retorna un valor diferente de la constante CONFIGURED_STATE, entonces no será posible realizar ninguna transacción, por lo cual la función *USBDeviceTasks()* debe volver a ejecutarse inmediatamente. Luego, la función *USBIsDeviceSuspended()* retorna un valor falso si es que el modulo USB no ha sido suspendido por alguna instrucción interna. Finalmente, la función *APP_DeviceCDCBasicDemoTasks()* realiza el proceso de comunicación mediante la clase CDC entre el PIC18F4550 y el *Host* cumpliendo con una serie de tareas, incluidas las rutinas de envío y recepción de tramas seriales asíncronas que serán generadas a través de la clase CDC. El archivo app_device_cdc_basic.c, cuya ruta se muestra en el gráfico 14.16, contiene a la función APP_*DeviceCDCBasicDemoTasks()*. En el programa 14.3 se observa el contenido de la función.

Programa 14.3. Estructura de la función APP_DeviceCDCBasicDemoTasks()

```
void APP_DeviceCDCBasicDemoTasks()
{
   if(BUTTON_IsPressed(BUTTON_DEVICE_CDC_BASIC_DEMO) == true)
  {
    if(buttonPressed == false)
    {
      if(mUSBUSARTIsTxTrfReady() == true)
      {
        putrsUSBUSART("Hola Mundo\n\r");
        buttonPressed = true;
      }
    }
  }
  else
  {
    buttonPressed = false;
  }
  if( USBUSARTIsTxTrfReady() == true)
  {
```

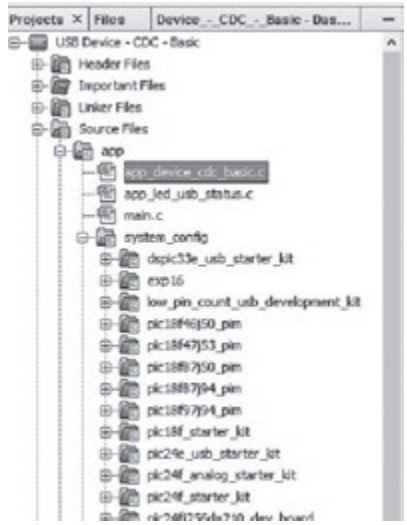
```
    uint8_t i;
    uint8_t numBytesRead;
    numBytesRead = getsUSBUSART(readBuffer, sizeof(readBuffer));
    for(i=0; i<numBytesRead; i++)
    {
      switch(readBuffer[i])
      {
        case 0x0A:
        case 0x0D:
          writeBuffer[i] = readBuffer[i];
          break;
        default:
          writeBuffer[i] = readBuffer[i] + 1;
          break;
      }
    }
    if(numBytesRead > 0)
    {
      putUSBUSART(writeBuffer,numBytesRead);
    }
  }
  CDCTxService();
}
```

Analizando el programa 14.3, se puede observar que dentro de la función se realiza un análisis del estado del pin del Puerto B (posteriormente, se mostrará cómo modificar los pines que serán evaluados en el programa). Si el pin se encuentra pulsado, y el *Buffer* de transmisión del USB está disponible, se enviará el mensaje "Hola Mundo\n\r", el cual podrá ser visualizado por un *software* de comunicación serial, tal como el Hyperterminal. Por otro lado, si se transmiten bytes desde el *software* al microcontrolador, la función *getsUSBUSART()* devolverá la cantidad de bytes recibidos en el *Buffer* de recepción. Como se observa en el bucle iterativo *for()*, si los bytes recibidos son 0x0A o 0x0D ('\n' o '\r'), el byte recibido se almacena en un vector de nombre *writeBuffer*; en cambio, para cualquier otro valor binario, el mismo vector recibirá el byte recibido más 1. Una vez generado el vector *writeBuffer* con los datos recibidos, se retransmitirá la información recibida haciendo uso de la función *putUSBUSART()* y se podrá observar en el Hyperterminal que los bytes ASCII transmitidos serán recibidos con la sumatoria de 1.

Finalmente, en los archivos buttons.c y leds.c se encuentra la definición de los pines que harán oscilar los LED y que serán utilizados como entradas para detectar el estado de los pulsadores. El contenido de estos archivos se puede adecuar, ya que, de no utilizar la tarjeta PICDEM FS USB, es necesario adecuar los estados de los LED y pulsadores a otros pines. En el gráfico 14.17 se observa la ubicación

de los archivos en el proyecto. En el programa 14.4 se muestra la nueva configuración del archivo buttons.c para acoplar los pulsadores del circuito esquemático del gráfico 14.14 al programa de ejemplo propuesto por la MLA. Finalmente, en el programa 14.5 se muestra la modificación del archivo leds.c para acomodar los pulsadores al mismo circuito esquemático.

Gráfico 14.16. Ruta de acceso para abrir el archivo app_device_cc_basic.c



Elaboración propia.

Gráfico 14.17. Ruta de acceso para abrir los archivos leds.c y buttons.c

Elaboración propia.

Programa 14.4. Modificación del archivo buttons.c

```
#define S2_PORT  PORTBbits.RB4
#define S3_PORT  PORTBbits.RB5

#define S2_TRIS  TRISBbits.TRISB4
#define S3_TRIS  TRISBbits.TRISB5
#define BUTTON_PRESSED     0
#define BUTTON_NOT_PRESSED  1

#define PIN_INPUT         1
#define PIN_OUTPUT      0

#define PIN_DIGITAL       1
#define PIN_ANALOG      0
```

Programa 14.5. Modificación del archivo leds.c

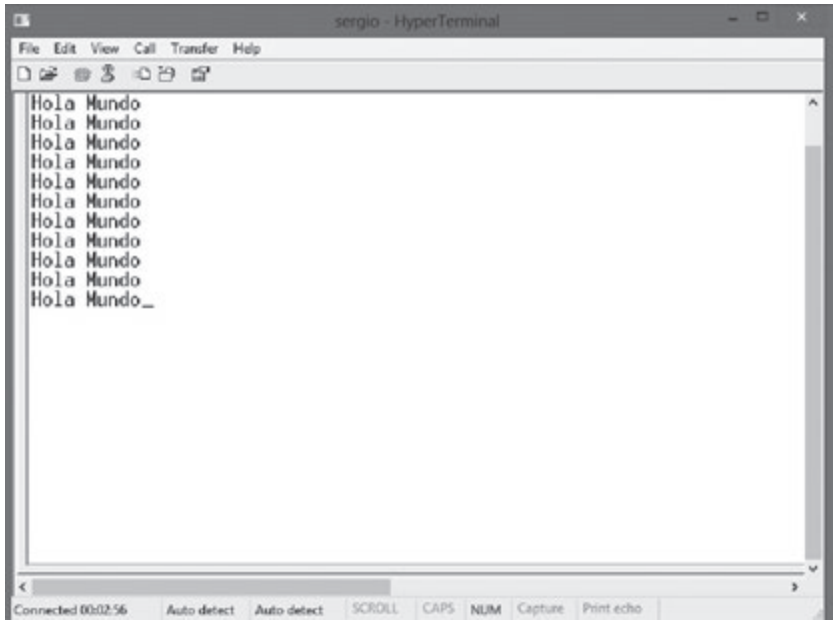
```
#include <stdbool.h>
#include <leds.h>
#include <xc.h>

#define LED_D1_LAT LATBbits.LATB0
#define LED_D2_LAT LATBbits.LATB1
#define LED_D3_LAT LATBbits.LATB2
#define LED_D4_LAT LATBbits.LATB3

#define LED_D1_TRIS TRISBbits.TRISB0
#define LED_D2_TRIS TRISBbits.TRISB1
#define LED_D3_TRIS TRISBbits.TRISB2
#define LED_D4_TRIS TRISBbits.TRISB3

#define LED_ON  1
#define LED_OFF 0
```

Una vez modificado el programa, es posible grabarlo en la memoria FLASH del PIC18F4550. Luego, se deberá conectar el computador con el conector USB tipo B del circuito del gráfico 14.14 mediante un cable USB tipo A-B y observar el reconocimiento del sistema operativo para el inicio de la enumeración. En el gráfico 14.18 se muestra el resultado de grabar el PIC18F4550 con el programa de ejemplo mostrado y conectarlo al computador a través de un cable USB. Una vez cargado el archivo mchpcdc.inf correspondiente para esta aplicación, el sistema operativo creará un puerto COM virtual (en el gráfico 14.18 se puede observar el puerto COM19). Posteriormente, el LED conectado al pin RB0 del PIC18F4550 comenzará a oscilar a una frecuencia visible, indicando que el proceso de enumeración y reconocimiento del sistema embebido ha sido satisfactorio. Una vez obtenida esta configuración, será posible abrir el Hyperterminal y probar la comunicación con el PIC18F4550, tal como se muestra en el gráfico 14.19.

Gráfico 14.18. Reconocimiento del sistema operativo del PIC18F4550 bajo el proyecto USB Device–CDC–Basic

Elaboración propia.

Gráfico 14.19. Prueba de funcionamiento en el *software* Hyperterminal



Elaboración propia.

Preguntas de repaso

1. ¿Cuál es la función de un transceptor USB?
2. Explique qué tarea cumple el *Hub* USB en el *Host*.
3. ¿En qué consiste el proceso de enumeración?
4. Indique dos diferencias entre el estándar USB 3.0 y el USB 2.0.
5. Explique la diferencia entre una transferencia y una transacción.
6. ¿Qué entiende por una clase USB?
7. ¿Cuántos pines posee un conector USB tipo B 2.0? ¿Cuáles son y qué funciones cumplen?
8. ¿Cuál es la función de un descriptor USB?
9. ¿Cuál es la función de las resistencias de *Pull Up* en las líneas D+ y D-?

Respuestas

Respuestas del capítulo 14:

http://bit.ly/1QmiXWA

Bibliografía

AXELSON, Jan (2007) Serial Port Complete Second Edition. Madison: Lakeview Research.

———— (2009) USB Complete Fourth Edition. Madison: Lakeview Research.

CADY, Fredrick (2006) Software and Hardware Engineering: Assembly and C Programming for the Freescale HCS12 Microcontroller. Estados Unidos: OUP.

CHU, Pong P. (2008) FPGA Prototyping by VHDL Examples, Xilinx Spartan-3 Version. Nueva Jersey: John Wiley & Sons.

DOGAN, Ibrahim (2008) Advanced PIC Microcontrollers Projects in C. From USB to RTOS with the PIC18F Series. Massachusetts: Elsevier.

———— (2014) PIC Microcontroller Projects in C. Basic to Advanced. Oxford: Elsevier.

EADY, Fred (2004) Networking and Internetworking with Microcontrollers. Burlington: Elsevier.

GALEANO, Gustavo (2009) Programación de sistemas embebidos en C. Bogotá: Alfaomega Colombiana.

GIDEKEL, Alan (2006) Introducción a la identificación por radio frecuencia RFID. Buenos Aires: Telectrónica Codificación.

HUDDLESTON, Creed (2007) Intelligent Sensor Design Using the Microchip dsPIC. Massachusetts: Elsevier.

JASIO, Lucio di (2008) Programming 32-bit Microcontrollers in C, Exploring the PIC32. Burlington: Elsevier.

———— (2014) Graphics, Touch, Sound and USB. User Interface Design for Embedded Applications. Estados Unidos: Lulu Enterprises.

MICROCHIP TECHNOLOGY (2008) PICkit 2 Programmer/Debugger User's Guide. (http://ww1.microchip.com/downloads/en/devicedoc/51553e.pdf)

———— (2009) PIC18F2455/2550/4455/4550 Data Sheet. (http://www.microchip.com/downloads/en/devicedoc/39632e.pdf)

PEREIRA, Fabio (2008) HCS08 Unleashed: Designer's Guide To the HCS08 Microcontrollers. Createspace.

SÁNCHEZ, Julio y CANTON, María P. (2007) Microcontroller Programming the Microchip PIC. Nueva York: CRC Press.

WEBSTER, John (1997) Design of Pulse Oximeters. Winsconsin: Taylor & Francis Group.

Recientes publicaciones de Editorial UPC

2015

Galdos, Gonzalo
Toma de decisiones. Elecciones acertadas para el éxito personal y profesional. Segunda edición

Lossio Chávez, Felix y Vargas Negrete María del Carmen
Sueños de oriente. Consumo de cine de la India y telenovelas coreanas en el Perú

Arnáez Braschi, Enrique
Enfoque práctico de la teoría de robots. Con aplicaciones en Matlab

Carrera de Diseño y Gestión en Moda
Técnicas de patronaje Tomo II - Hombre

Herz Ghersi, Jeannette
Apuntes de Contabilidad financiera. Segunda edición

Aguirre, Mauricio; Maldonado, Claudia; Peña, Cinthia, y Rider, Carlos (comps).
Cómo leer y escribir en la universidad. Prácticas letradas exitosas. Segunda edición

Aguirre, Mauricio; Maldonado, Claudia; Peña, Cinthia, y Rider, Carlos (comps).
Cómo leer y escribir en la universidad. Cuaderno de trabajo. Segunda edición

2014

Olins, Wally
Brand New. La esencia de las futuras marcas

Velásquez Aguilar, Luis Oscar
Niños hospitalizados. Guía de intervención psicológica en pacientes infantiles

Peña Larrea, Cinthia (Compiladora)
Más allá de las palabras. Una propuesta de análisis crítico del discurso

Soria Aguilar, Alfredo F. y Osterling Letts, Madeleine
Contratos modernos. Elementos esenciales y reglas aplicables para acuerdos comerciales

Rodríguez Félix, César y Mauricio Alza, Saby
Nutrición oncológica. Guía de alimentación para vivir mejor

Chu Rubio, Manuel y Agüero Olivos, Carlos
Matemática para las decisiones financieras

Romero Tapia, José Fernando
Integrando mercados. MILA, motor de la Alianza del Futuro

Arrascue Córdova, Lily
Física mecánica. Nivelación para estudiantes universitarios

Arnáez Braschi, Enrique
Enfoque práctico del control moderno. Con aplicaciones en Matlab

Sardón Taboada, José Luis (editor)
Revista de Economía y Derecho
Verano de 2014, vol. 11, nro. 41

Valdivia Pareja, Álvaro
Suicidología. Prevención, tratamiento psicológico e investigación de procesos suicidas

Egoavil Vera, Juan Raúl
Fundamentos de matemática. Introducción al nivel universitario

Chu Rubio, Manuel
Finanzas para no financieros. Cuarta edición

Parodi Revoredo, Daniel
Conflicto y reconciliación. El litigio del Perú contra Chile en la Corte de La Haya (2008-2014)

Aguirre, Mauricio; Maldonado, Claudia; Peña, Cinthia, y Rider, Carlos (comps).
Cómo leer y escribir en la universidad. Prácticas letradas exitosas

Aguirre, Mauricio; Maldonado, Claudia; Peña, Cinthia, y Rider, Carlos (comps).
Cómo leer y escribir en la universidad. Cuaderno de trabajo

Encuentre más publicaciones de Editorial UPC,
en versión impresa y digital, ingresando a:
www.upc.edu.pe/editorialupc

Visite la página de Facebook Editorial UPC:
www.facebook.com/editorialupc

Otros títulos publicados por Editorial UPC

Enfoque práctico de la teoría de robots.
Con aplicaciones en Matlab
Enrique Arnaez Braschi

Enfoque práctico del control moderno.
Con aplicaciones en Matlab
Enrique Arnaez Braschi

Física mecánica.
Nivelación para estudiantes universitarios
Lily Arrascue Córdova

Avances en Neurociencias: Neuropéptidos
Investigación básica y clínica
Rafael Coveñas y Luis Aguilar

Encuentre más publicaciones de Editorial UPC,
en versión impresa y digital, ingresando a:
www.upc.edu.pe/editorialupc
Visite la página de Facebook Editorial UPC:
www.facebook.com/editorialupc

EDITORIAL UPC

La publicación es una guía teórica y práctica para el aprendizaje de la metodología de programación y diseño de aplicaciones con sistemas embebidos de 8 bits de CPU, específicamente tomando como modelo el microcontrolador PIC18F4550 de la marca Microchip®.

El libro trata de enfocar aquellos aspectos que son de suma importancia conocer para el diseño y desarrollo de hardware basado en sistemas embebidos: entender la arquitectura del microcontrolador, conocer las herramientas de desarrollo disponibles en el mercado y la Internet para el diseño de firmware, dominio del lenguaje ensamblador y dominio de un lenguaje de alto nivel como el ANSI C. También se busca explicar el funcionamiento de los módulos periféricos internos del microcontrolador y presentar ejemplos de diseño de hardware y firmware de diversas aplicaciones reales.

Sergio Salas, ingeniero electrónico y autor de esta publicación, propone el uso de dos herramientas de aprendizaje: el hardware PIC18F4550, sobre el cual se presentan diversas aplicaciones reales de diseño electrónico digital, las cuales el lector las puede tomar como referencia para implementarlas en un proyecto personal similar u otro más grande o simplemente para verificar su funcionamiento. La segunda herramienta a utilizarse es el software MPLAB X IDE de la marca Microchip® que es la interfaz de desarrollo de firmware.

Escanear este código
con tu smartphone